O'REILLY®

ActivityPub

Programming for the Social Web

Evan Prodromou

Praise for *ActivityPub*

ActivityPub represents the best hope we have to disenshittify the internet we're stuck with today, and to build a new, good internet that is both enshittification-resistant and accessible to the normies we love and want to protect from the insatiable, predatory horniness that tech bros have for enshittifying the services we depend on.

—Cory Doctorow, author of
Little Brother *and* The Internet Con

When Threads started our ActivityPub implementation, we had to map all the high-level concepts of the protocol onto our internal infrastructure while also making sure we respected the nuanced implementation details. Evan's book elegantly covers both ends of this spectrum with grace and compelling context. I highly recommend it!

—Peter Cottle, Software Engineering Director, Meta Threads

The ActivityPub protocol is the foundation of the modern social web, uniting Mastodon, Flipboard, WordPress and many more platforms into one social network. For the first time ever, starting your own social app does not require you to fight against network effects. Evan's book will accurately guide you through the intricacies of implementing this powerful protocol in your own social app.

—Eugen Rochko, creator and founder of Mastodon

ActivityPub is one of the most important technologies remaking the modern internet, and this book is full of the kind of insights that only Evan Prodromou, as a pioneer of ActivityPub tech, could provide.

—Anil Dash, VP of Developer Experience, Fastly,
and CEO, Glitch

Social networks have consolidated into a handful of platforms run by large corporations, controlling users' data and every aspect of their participation on the social web. What makes ActivityPub–and this book–so crucial is that it takes that power and control over social interactions and hands it back to the users.

—Jessica Tallon, ActivityPub coauthor and Founding Technologist, Spritely Networked Communities Institute

As the creator of the ActivityStreams initiative in 2009, I envisioned a standard that would surpass syndication formats like RSS to ensure long-term competition and innovation across the social web. Evan Prodromou has moved beyond this vision and, through relentless effort, cat-herded it into a robust, popular open standard. His clear explanations and rigorous approach make nuanced concepts accessible, enabling you to build compelling, interconnected experiences for the modern social web. This book represents a milestone in the evolution of the open, social web.

—Chris Messina, inventor of the hashtag

Social networking is increasingly how our species talks to itself–yet each new offering trying to be the standalone town square and profit center hasn't worked. We need social networking that is decentralized, federated, and humanized, not owned by anyone. ActivityPub seems the technology most likely to get us there. This book will help.

—Tim Bray, editor of the Internet Standard specifications for JSON and XML

The ActivityPub protocol is the most significant advancement to the open web in 30 years. It standardizes human connection and liberates people from today's social media walled gardens. This developer's guide will help you build entirely new ways to connect people and content on the internet. The world awaits your big idea.

—Mike McCue, CEO and cofounder, Flipboard

ActivityPub is about to democratize the social web! It may be the biggest innovation since Web 2.0, offering a paradigm shift that could completely redefine how we connect and share online. However, with its flexibility comes the challenge of diverse implementations, making this book an essential guide to ensure interoperability and safeguard the future of the fediverse.

—Matthias Pfefferle, Open Web Lead, Automattic

Learn everything you need to know about the technology that's shaking up the social media world and helping to build a less centralized, more inclusive web. Evan knows what he's talking about, and getting to learn ActivityPub from one of its creators is a rare gift.

—Manu Sporny, Founder and CEO, Digital Bazaar

ActivityPub makes decentralized social networks happen, and who better to learn from than Evan Prodromou, the founding architect of ActivityPub?

—Christine Lemmer-Webber, ActivityPub coauthor/coeditor

Evan's book demystifies ActivityPub, bringing it down from the web standards pedestal to make it a practical tool in any developer's toolbox. It's just what we need for ActivityPub to really level the playing field for developing social web applications and services.

—Amy Guy (rhiaro), coeditor of ActivityPub

Evan uses his unique expertise and personality to make *ActivityPub* an enjoyable and informative read. Developers will find the structure accessible and the prose clear and light.

—Mallory Knodel, Social Web Foundation, author of How the Internet Really Works

ActivityPub has needed a comprehensive and authoritative O'Reilly book to help drive interop for all the new microblogging systems—finally, it's here!

—Dave Winer, developer of web formats and protocols

A practical and clear roadmap to building your first fediverse server—that happens to mark many of the potholes you're bound to encounter on the way there.

—Darius Kazemi, author of the express-activitypub server

ActivityPub

Programming for the Social Web

Evan Prodromou

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

ActivityPub
by Evan Prodromou

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Amanda Quinn
Development Editor: Sarah Grey
Production Editor: Elizabeth Faerm
Copyeditor: Sharon Wilkey
Proofreader: Kim Cofer
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

September 2024: First Edition

Revision History for the First Edition
2024-09-20: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098162740* for release details.

978-1-098-16946-6

"This document is dedicated to all citizens of planet Earth.

You deserve freedom of communication; we hope we have contributed in some part, however small, towards that goal and right."

—The ActivityPub Specification (2018)

Table of Contents

Preface

ActivityPub is the standard API and protocol for social networks. It is one of the most exciting areas in software development today, providing a way of connecting social networks together into a single, integrated social web.

I've been lucky enough to work on ActivityPub, and its predecessors, for more than 15 years. As cochair of the W3C working group that created the standard, and coeditor of the specification document, I have been intimately involved in ActivityPub's development. I'm so honored to get a chance to introduce you to its power and elegance. I hope you'll find it as delightful as I do.

Audience

This book is primarily for software developers interested in creating ActivityPub client or server software. It could also be useful for software architects or designers who are considering how to structure an ActivityPub-centric project.

Chapters 1 and 6 cover ActivityPub's past, present, and future. They are mostly narrative, without code or data examples, and may be interesting for policy makers, business leaders, and entrepreneurs who are investigating how to use ActivityPub-enabled software in their organizations.

Prerequisites

This book uses many examples in JavaScript Object Notation (JSON), JavaScript, and Python. Each language gets a brief description when it is introduced, but familiarity with JSON as a data format, and JavaScript and Python as programming languages, will be a big help in understanding the examples. However, I try not to use too many fancy features of each language, so if you're familiar with any modern programming language, you should be able to puzzle out the examples.

It can help to be familiar with the architecture of the web, including how Hypertext Transfer Protocol (HTTP) works and how Hypertext Markup Language (HTML) pages are developed.

Experience with application programming interfaces (APIs), especially APIs for social networks like X or Facebook, can be helpful to understand the motivation for some examples but isn't strictly required.

Structure

Chapter 1, "Welcome to the Fediverse" introduces social software and the motivations for the creation of ActivityPub.

Chapter 2, "Activity Streams 2.0" covers Activity Streams 2.0 (AS2), the social data standard. The different types and properties of AS2 objects are discussed, as well as how AS2 objects can be processed in different programming languages.

Chapter 3, "The ActivityPub API" introduces the ActivityPub API, which connects clients to servers. Through an extended example, it shows how API clients can be developed, and how they can create and interact with social content.

Chapter 4, "The ActivityPub Protocol" covers the ActivityPub federation protocol, which connects servers to other servers. An example for creating automated "bot" accounts is provided to focus on server interactions.

Chapter 5, "Extending ActivityPub" covers extensions to AS2 and ActivityPub. It discusses how compliant programs handle extension data and gives ideas of how to create new extensions.

Chapter 6, "Far Horizons" lays out a roadmap for ActivityPub into the future. If you have any questions about where this technology is going, this is a great place to look.

Appendix A, "Activity Vocabulary Types" is a reference covering the types in the Activity Vocabulary.

Appendix B, "Activity Streams 2.0 Properties" is a parallel reference, providing similar coverage for the properties in the Activity Vocabulary.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/evanp/activitypub-book*.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*ActivityPub* by Evan Prodromou (O'Reilly). Copyright 2024, 3102451 Nova Scotia Company, 978-1-098-16946-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://www.oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/activitypub*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

So many people made this book possible. I cannot even begin to thank them all, but I'm going to try to name a few of the most important ones.

Thanks in no particular order to Jon Postel, Dave Winer, Sir Tim Berners-Lee, Molly Holzschlag, Tantek Çelik, Blaine Cook, Chris Messina, David Recordon, Joseph Smarr, Brad Fitzpatrick, John Panzer, Mónica Goren, Will Norris, Aaron Parecki, Ben Werdmuller, Bob Wyman, Henry Story, Jon Phillips, Kevin Marks, Brett Slatkin, Julien Genestoux, Mike Macgirvin, Matt Lee, Mikael Nordfeldth, Evan "Rabble"

Henshaw-Plath, Kellan Elliott-McCrea, Kaliya "Identity Woman" Hamlin, Dewitt Clinton, Eran Hammer, Al3x Payne, Ed Finkler, AJ Jordan, Nathan Schneider, Ilya Zhitomirskiy, Dan Grippi, Max Salzberg, and Raphael Sofaer, Dan Brickley, Steve Ivy, Tim Bray, Ian Forrester, and Mike McCue.

Thank you to everyone who attended the Federated Social Web summits.

Thank you to Ward Cunningham.

Thanks to the millions of users of identi.ca. Thank you to Brooke Vibber, James Walker, and Zach Copley, and everyone else who worked on OStatus and StatusNet.

Thanks to my coeditor on the AS2 specification, James Snell, and to the coeditors of ActivityPub, Christine Lemmer-Webber, Jessica Tallon, Erin Shepherd, and Amy Guy. Thanks to Arnaud Le Hors, Bebe Roberts, Ben Goering, Sandro Hawke, Harry Halpin, Ann Bassetti, and everyone on the Social Web Working Group at the W3C. It was an honor to work with you all; we made something that mattered together.

Thank you to Eugen "Gargron" Rochko.

Thank you to Darius Kazemi.

Thank you to Ryan Barrett, Dmitri Zagdulin, Juan Caballero, Johannes Ernst, Philippe Le Hégaret, Matthias Pfefferle, and everyone else in the SocialCG at the W3C.

Thank you to everyone who shares their thoughts, their creations, and their life sincerely on the fediverse, who encourages people to join, and who welcomes those who do.

Thank you to my fellow cooperative members at CoSocial.ca.

Thank you to my cofounders at the Social Web Foundation, Mallory Knodel and Tom Coates, all our advisors, and all the stakeholders who have supported us.

Thank you to my technical reviewers (they're all named above!).

Thank you to Sarah Grey, Amanda Quinn, Liz Faerm, Mike Loukides, and everyone at O'Reilly Media.

Thank you to my colleagues at the Open Earth Foundation.

Thank you to my parents, Stav and Ami; and to my brothers, Andy, Ted, and Nate, and their families.

Most of all, thank you to my wife, Michele Ann Jenkins, who had the idea for this book, to my daughter Amita June, and to my son Stavro, for having patience with me while I wrote this book. I love you all.

CHAPTER 1

Welcome to the Fediverse

I'm so glad that I get to be the person to welcome you to the social web. I'd like to use this opportunity to explain why we're here, what we're doing, and maybe where you can fit in. Let's start with the first steps: understanding social networks.

What Social Networks Do

What is a social network? At its most basic, a *network* is a collection of objects with distinct connections between them. A railway network, for example, is a set of railway stations connected by train tracks. A computer network is a potentially very big set of computers, connected by wires or electromagnetic waves to one another.

A *social network* is a group of people and the connections they form with one another. Like computer networks or television networks, information travels along these connections.

People often talk about social networks as the apps and websites that model our social connections and convey information that we send, although that's probably better thought of as *social-network software*, *social-network services*, or just *social software*. I'll be using *social network* to mean either the people or the software interchangeably in this book, but I'll try to be clear when I mean one or the other.

As of 2024, as I write this book, social-network services have become an integral part of modern society.[1] We use them for our intimate social connections, for our civic and political life, for entertainment, for news. We use them to hold our memories, maintain contact with friends, and share important milestones.

1 Future humans: if this is no longer the case in your time, thank you for reading this historical text.

Because social-network services are so ubiquitous, it can be hard to remember that the underlying model is so simple. Many of the most interesting innovations in consumer technology over the last decade have been attached to social networks to take advantage of features like identity and content distribution. Some of my photo-sharing services let me add an animated sticker or a thought balloon to my party photos; the developers who created that feature attached it to a social network to benefit from the distribution.

Other networks let you reuse your identity to make video calls or log in to accounting websites. The identity structure of social networks is so robust that other services can depend on them for registration and login. Figure 1-1 shows the features of a typical social-network platform.

We do a lot with social software, and a lot of websites and apps call themselves social-network services. Here are some of the commonalities:

An identity
: In social software, you usually have a unique identity. Sometimes this is explicit, like a unique username or handle. Other times, it's implicit—even though we never see the identifier, we know that our account is unique.

A profile
: The profile is your home base on the social network; it contains information that other people can see about you. It might include your name (or a pseudonym), location, age, hobbies, links to other profiles on other social networks, an avatar picture, a funny or serious self-description, and maybe other profile properties.

Relationships
: Social software is social because it supports connections among users. This is how you tell the software and other people on the service about people who matter to you. We make connections with friends, family, colleagues from work, neighbors, celebrities, politicians, or just cool people we met online who we want to keep in touch with. Sometimes these relationships are *bidirectional*, meaning both people are actively part of the relationship (these are sometimes called *friends*, although that doesn't necessarily mean a real-life friend). And sometimes the relationships are *unidirectional*, meaning one person is interested in the relationship and the other one is mostly unaware of it. These are sometimes called *followers*/*following* relationships or *fan* relationships.

Groups

Along with having relationships with individual people, we can become members of groups. Sometimes everyone knows one another in the group, like a family or a city block; other times the relationship is looser, like all the fans of a TV show, worldwide.

Posts of new content

Whether it's bookmarks, short texts, long texts, photos, drawings, or videos, social networks let us share things we make with people we know and care about. Sometimes we post a thought that goes out to everyone we know; other times it goes only to a particular person or group.

A home timeline

Social networks often notify you of all the new content from people who you have a relationship with or groups that you're a member of. That all comes into a single stream of activities that you can read. Sometimes they're sorted with the newest on top, and sometimes they're sorted with the most interesting on top. Either way, you can scroll through the feed to see what's happening in your world right now.

Reactions and replies

When people we know show us something about their lives, we want to tell them or show them that it matters to us. We might comment on a photo they posted, "like" it, or give an emoji reaction. As important as the original content can be, sometimes the reactions and replies are almost as interesting and as good a place to connect.

Reposting

Sometimes a connection posts a question or request that we can't help with ourselves, but someone we know might be able to. Other times, an image or a video makes a point in a way that we can't put into words. We want all our connections to see it, and maybe experience the same feeling, and maybe spread their post a little further. By reposting, we can use our network to give content we like a little more attention; the things we repost go out to all our connections in the same way that new things we make do.

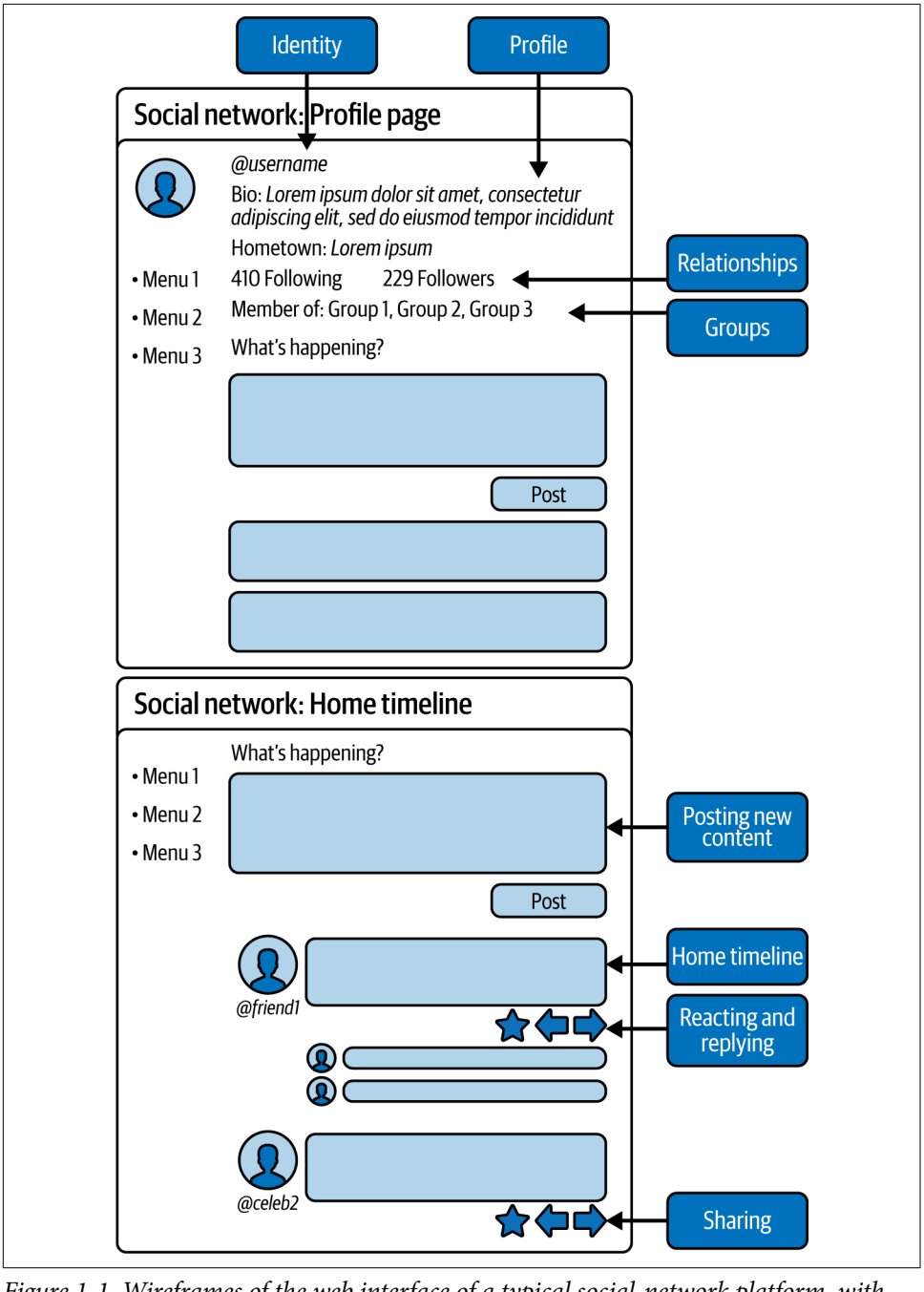

Figure 1-1. Wireframes of the web interface of a typical social-network platform, with abstract features highlighted

What Social Networks Don't Do

People today use social networks so much in their online lives, it can be hard to remember what social networks *don't* do. In fact, we've gotten so used to the "normal" way that social-network software can work that we have a hard time understanding any limitations. On the major social networks, you can't always do the following:

Connect with someone on another network
: Not all your friends use every new social network. Maybe you want to stay involved with people who haven't updated to the cool new app you're using. People sometimes get comfortable in the apps they know and don't want to jump into a new social network every few months. It would be nice to let your friends stay where they are, but be able to share content from your new app with them, and maybe even see when they respond to it.

See content in your home feed from other networks
: On the flip side, when people important to you are developing new ideas or new media on other services, it'd be nice to see that stuff in your preferred home feed (which works just the way you like it) instead of having to download a new app, sign up, agree to terms and conditions, and reconnect to whichever friends are already on the app.

Move your account from one network to another
: Many services let you download your files and even some of your connections in a ZIP file. Some ambitious smaller services can even import that ZIP file to a new account. But you can't rewire your friendships, group memberships, or other relationships to the new account.

Post audio on a photo service
: A lot of social-network software is oriented toward a single kind of content: just photo sharing, just videos, just audio, just short text, or just long text. You can sometimes tweak the service to get around these restrictions—for example, by screenshotting a long text and sharing it as an image, or playing music in a video over still images—but the process is usually clumsy and complicated. Often, what matters to us is the community, and our social relationships on the service, and not the particular media type it was started with.

Decide who can see your content and interact with you
: All social-network services have privacy policies and security settings, but they tend to be one-size-fits-all. If you have particular needs—like sharing a surprise party invitation with all your friends except the guest of honor—figuring out how to do that can be difficult. And if you're having problems with a person or a group of people, and you want to prevent them from bothering you and your friends, you have to provide a report to a busy Trust and Safety team, which may

not share your priorities and concerns. The team might even delegate these tasks to AI assistants that don't empathize with personal feelings.

Choose your own business model
: All social-network services cost money to run, and eventually all of them look for a way to make money. They may land on subscriptions, premium services, advertising, or a cooperative model. But whatever business model the network service chooses, that's the one you're stuck with. You can't decide you want to pay a subscription and see no ads, or that you're willing to volunteer to offset your monthly charges. If you don't like the model, you have to choose between living with how the business works and staying connected to your friends, family, and colleagues, or giving up on the network and getting disconnected.

Make up new ways of interacting with people you care about
: Many social-network services are built on a gimmick for how to interact: they might, say, limit the number of characters in a text message to 140 characters or allow only a small number of connections. These constraints can be great incentives for creativity, reflection, and deeper social connection. Given how important they can be, it's strange that your ability to experiment with different constraints, or expanded abilities, is often deeply curtailed.

Sift, sort, and search data your own way
: You may want to find only people in your social network who are interested in dating someone like you. Or you might want to get introduced to a random friend-of-a-friend on a daily basis. You might want your work updates to come before your news, which comes before personal content, which comes before entertainment—or exactly the reverse. Regardless, the business models of many social-network services depend on showing you content in a certain order and hiding other content from you entirely. Using their application programming interfaces (APIs) or data feeds to show things in different ways is usually prohibited by terms of service.

Save your content and community after the service shuts down
: One of the most difficult social-network experiences is the service shutdown. Sometimes a venture-capital-funded startup is willing to spend money to get lots of users for a new service, in hopes of later recovering that money as revenue for a future business model. Other times, a major corporation launches the service as an experiment or side project, then trims it to boost share prices. Regardless, when a service shuts down, the entire universe inside that service collapses: all the content disappears, and all the human connections are erased. Even if you manage to get a backup of your personal content, republishing it is difficult, and your connections are gone for good.

All these features leave gaps between our social networks, making them islands on the internet. Sometimes those islands are *really* big and feel more like continents unto themselves. But they remain isolated and disconnected from one another.

People do what they can to work around this disconnection, of course. Sometimes you'll see a friend post a screenshot of a different app into their feed, since that's the only way they can share it. Occasionally you'll see content captured in a screenshot in one app, posted to another app, and then captured again to be posted to yet another social network!

Some tools let you do a little bit of transferring content between networks. Some networks will use another service's API to push updates through to the other network, but any feedback (such as replies or likes) from users of that service rarely occurs. On Apple iOS and Android phones, you can "share" a picture from one network to another too.

Locked In to Social Networks

Overall, social-network services can feel really frustrating to a lot of their users, for a thousand reasons. So, the big question is: why don't they just leave?

One of the awe-inspiring powers of social networks is encoded in a principle called *Metcalfe's law*. Bob Metcalfe, cofounder of computer-networking-hardware company 3Com Corporation, observed this property of communications networks in 1983. He noticed that if a network has N nodes, each node can contact $N - 1$ other nodes, so there are $(N \times (N - 1)) / 2$, or about N^2 possible connections. Since the value of the network is in its connections, he concluded that the value of a network to its users goes up by the *square* of the number of users. In other words, big networks are much, much more valuable than small networks, as illustrated in Figure 1-2.

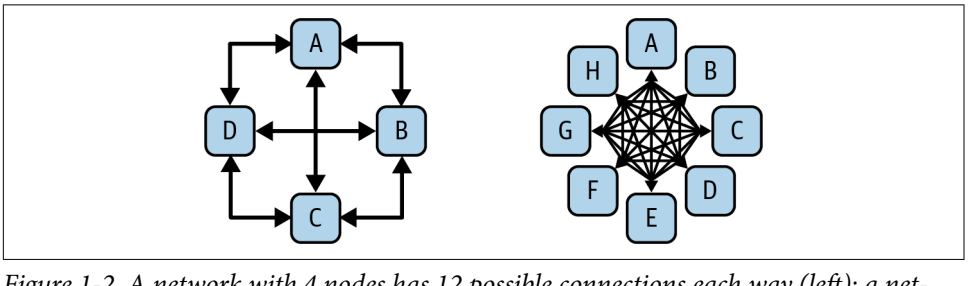

Figure 1-2. A network with 4 nodes has 12 possible connections each way (left); a network with 8 nodes has 56 possible connections (right)—it's denser, more active, and just more interesting than the smaller network

The value of big networks can have a positive effect; all of us who use the internet or the World Wide Web owe its vast utility to Metcalfe's law. But it makes it hard for small networks to get started. Users leave the small networks to join the big networks

since, by the law, they're much, much better. Everyone you know is there; everything you want to do is happening there. But users also have a hard time leaving the big networks to go to the small networks. They're much, much less fun.

Metcalfe's law explains why users get locked into a network. For those of us who feel frustrated with the social-network services we use, and want to try something new, Metcalfe's law is what keeps us going back to our old habits.

From Social Network to Social Web

Even if you love the experience of using social-network software, the series of disconnected islands that are our social-network services can be a real source of frustration. So what can you do? The model that some social-network developers have adopted is a *federated social network*: an alliance of social-network services connected through open standard protocols.

One good way to imagine this is by considering internet email. It doesn't matter what software you use for reading your email, or even what server your ISP or work organization uses. You can send email to anyone on the internet because your mail server is connected, via a complicated set of open standards, with every other mail server on the planet. Some email addresses aren't even for people; they could be mailing lists, bots, or other automated tools. The rules on your server don't matter to anyone else—for example, if your work uses virus scanners on incoming attachments or limits the size of files you can send. The rules your server sets up are right for you and your team, and nobody else gets a say in them.

Now consider the World Wide Web. Hundreds of millions of websites are on the internet today—each with its own menagerie of server software, programming languages, databases, file formats, network structures, APIs, and data feeds. But every one of those sites supports a simple (well, *kinda* simple) set of standards—like HTTP, HTML5, CSS, and JavaScript—to give users a rich, fascinating experience in the browser. And with links, browser users can move effortlessly between websites without anyone locking them in.

The *social web* applies the idea of internetworking to the realm of social software. It is a collection of social networks that allows users to make connections across network boundaries. A user on one network can follow a user on a different network and get the same updates in their home feed, as if they were on the same network. On the social web, people can post images, videos, and short and long texts; they can comment, like, and share; they can make jokes, have difficult discussions, and remember what matters most in life. Figure 1-3 shows how a siloed social network works: all clients use a single server, which holds all user accounts. The social web is like Figure 1-4: user accounts are on different servers, clients connect to the server for their user, and servers use an open standard to share information between them.

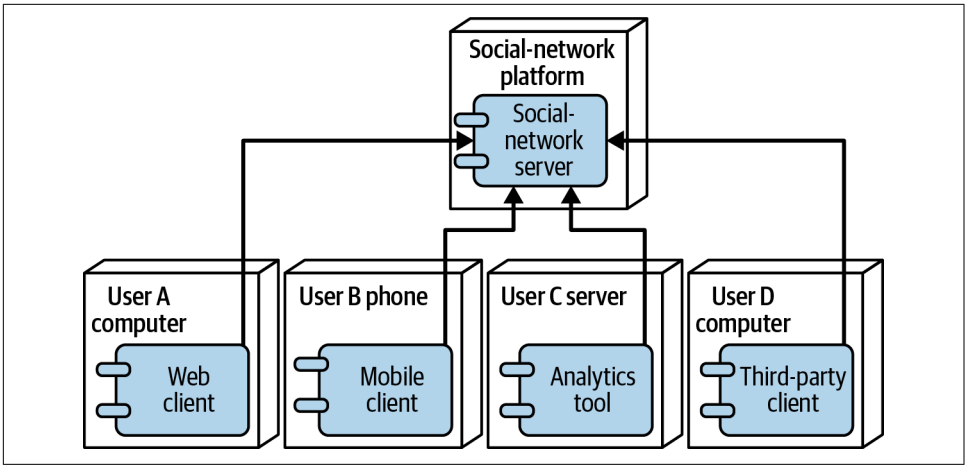

Figure 1-3. In a siloed social network, client software all connects to a single platform

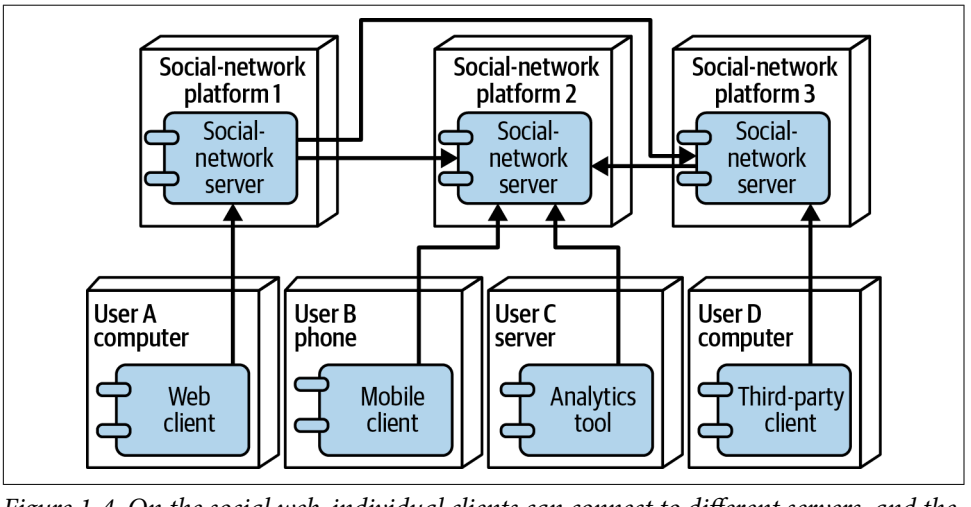

Figure 1-4. On the social web, individual clients can connect to different servers, and the servers use a backend protocol to connect people across networks

As of this writing, tens of millions of users are on the social web. And because anyone can join, there are tens of thousands of social-network servers—some with hundreds of thousands of users, others with only a single person. Some are commercial ventures; others are run by volunteers; still others are cooperatives or nonprofits. Many of these networks run open source software like Mastodon, WordPress, Pleroma, or Misskey and connect together with open standards.

The social web is an admission that one-size-does-not-fit-all. Every person relates to their social connections differently—with different needs, different processes, different

incentives. The diversity of software, both servers and clients, gives people the choices they need to construct their own social experience. Some people are going to find the most popular service, sign up for an account, and use the official client app for their entire time on the social web.[2] Others will shop around for the right server, the right community, and the right client apps, and then build scripts, connect services, fine-tune, tinker, connect, and generally get their social experience Just So.

The social web is the antidote to Metcalfe's law. Each of these social-network services is relatively small on its own, which should make them unviable. But, because they interconnect with so many other services around the world, the total number of possible connections is enormous. Your small family social-network service might serve only a handful of people from your household, but by connecting it with the rest of the fediverse, you gain a huge pool of potential friends to talk with.

The social web is sometimes called the *fediverse*, which is a portmanteau of *federated* and *universe. Federated* here refers to the independent services that can become part of the network whenever they choose, as long as they implement the necessary technical standards (more on those in the next section). Nobody can kick you off the fediverse; as long as your social network implements social standards, you're part of the social web. I use the terms *fediverse* and *social web* interchangeably in this book, which may get a little confusing, but I'm sure you're up for it.

The goal of this book is to teach you how to use the standards of the social web to make cool software and help human beings connect in fascinating ways. You can use social web technologies to connect an existing project to the social web, or you can start something new from scratch. You can make a tool that only a few people will use or something that changes the way people live their lives, online and off.

A Tour of the Standards

Let's take a quick look at the main relevant standards for social web software: Activity Streams 2.0 for data, the ActivityPub API for client-to-server programming, and the ActivityPub federation protocol for server interconnection. (Don't worry; they all get their own deep dives in later chapters.)

Activity Streams 2.0

Activity Streams 2.0 (*AS2*) is the data format for the social web. It's the common language that all fediverse software speaks: a standard JavaScript Object Notation (JSON) format for encoding information about common social-network entities.

2 Looking at you, mastodon.social users.

AS2 was designed to allow the maximum expressiveness possible, so you can take content, actors, and experiences from the many kinds of social-network services and share them across service boundaries.

AS2's data types include the following:

Actors
: People, groups, organizations, services, and applications. The active parts of a social network; those who *act*. Unrelated to Hollywood celebrities!

Objects
: Files like images, video, and audio; short and long texts; albums and lists. These are the stuff of websites and apps: the things we make, share, and do. More abstract types also exist, like places, relationships, events, and questions.

Activities
: AS2 has activity types for creating, reading, updating, and deleting objects—the *CRUD* necessary for making and maintaining files. It also includes activity types for reacting to others' work by liking or sharing it and for forming and changing the social graph by following people and joining groups.

In the language of the social web, an *activity* works like a sentence. An activity says that *someone* (the actor) *did something* (the activity) *to something else* (the object). Sharing activities is the most important part of connecting on the social web.

Each of the AS2 data types has dozens of properties that can be used to describe it: names, IDs, summaries, images, latitudes and longitudes and altitudes, heights and widths. The format is extremely *flexible*; you can include as much or as little property information about an object as you want. This flexibility lets us use the format for tons of applications. It can be a little harrowing getting AS2 data on the wire; you never know if you're going to get a big, bushy tree of data or a minimal, terse little packet. But once you get used to the format, you can appreciate the power this flexibility brings.

Almost as important, AS2 is *extensible*. If you come up with new kinds of content or activities or properties of things that were never dreamed of by the standards group that made AS2, you can define it and include it in your existing AS2 objects.

A lot more details about AS2 are in Chapter 2 and throughout the book. The appendices have detailed reference information on all the types and properties defined in the standard. It's really the glue that holds the social web together.

The ActivityPub API

The *ActivityPub API* is the mechanism for connecting web and mobile client software to ActivityPub servers. It's the entryway for humans to interact with the social web.

The ActivityPub API is a *standardized API*. Different servers provide the same interface; clients can use that interface on each service without knowing which service they're talking to. This is an unusual setup in the world of APIs; although some well-known and interesting patterns exist, like REST or GraphQL, not many fully standardized APIs are available.

As I've mentioned, all the interesting objects in the network have their own URLs that provide an AS2 representation of the object. This includes a number of interesting *feeds*, like an actor's *outbox* (a feed of all their activities) and *inbox* (a feed of all the activities of all the people they follow), and their *social graph* (people, apps, groups, and services they are connected to).

Client software can also use the API to create and share new activities. The primary mechanism for interacting with the entire fediverse is through the ActivityPub API, posting new activities to a particular endpoint.

The ActivityPub API specification defines what happens when each kind of activity is sent to the server—`Create`, `Like`, `Announce`, `Delete`, and so on. The server is responsible for implementing them and distributing them.

The ActivityPub API is extensible because the AS2 format is extensible. Developers interested in collaborating can define different kinds of objects, activities, and properties, and even different kinds of feeds.

One interesting part of the ActivityPub API is that you don't have to think about federation when you use it. That's all handled by the server software; as far as the client software is concerned, it's all one big network. In fact, the advantages of a standard social-network API are still pretty valuable. I'll cover the ActivityPub API in excruciating detail in Chapter 3. It's one of my favorite parts of the social web standard suite, and I hope you'll find it as mysterious and powerful as I do.

The ActivityPub Protocol

The *ActivityPub protocol* is the main tool for moving data across the social web. Chapter 4 covers the ActivityPub protocol in all its glory, so this is just a brief introduction. The protocol uses a similar mechanism to the ActivityPub API, but instead of transferring activities from the client to the server, it is a server-to-server protocol that allows one social network to send data to one or more other social networks, standardized by the World Wide Web Consortium (W3C), the standards body that manages specifications for many parts of the web stack. These internetwork connections are what allow the social web to grow.

If you're familiar with how internet email works, you can think of the ActivityPub federation protocol as the Simple Mail Transfer Protocol (SMTP) happening between email servers and the ActivityPub API as the Internet Message Access Protocol (IMAP) happening between the client and its own server. This analogy isn't perfect, but it points you in the right direction.

The ActivityPub protocol defines a set of important patterns that servers need to follow if they want to connect. Every object that they have on their service needs to be available through a web URL in the AS2 format. People, images, groups, replies—the whole thing is on the web.

The protocol defines a mechanism for delivering activities from one service to another. It also says what should happen when a service receives different kinds of activities: a `Follow` activity should make a connection between two actors. A `Like` activity should increase the number of likes on a video. Each activity does something different.

Distribution is a big part of the ActivityPub protocol. As an actor does various activities, those AS2 objects get distributed to everyone the actor wants to see them—often, all their followers. With social-network accounts that have tens of thousands, hundreds of thousands, or even millions of followers, distributing activities efficiently becomes a big deal.

If this sounds a lot like the ActivityPub API, well, that's no accident. The two halves of the ActivityPub specification were designed to work together seamlessly, so the structure, data formats, and interaction model are very much aligned.

I might sound like a broken record at this point, but because AS2 is extensible, so is the ActivityPub protocol. You can define new kinds of activities, and other services on the network that implement the extension can receive and apply those activities.

Together, these three standards form the skeleton of the social web. They connect client software with server software, and server software with other server software—all speaking a common language to describe social objects and interactions.

Of course, AS2 and ActivityPub are built on and interact with other standards: JSON and JSON-LD for data-transfer encoding; WebFinger for identity; OAuth 2.0 and HTTP Signatures for security; HTML for formatted content; a half dozen image, audio, and video file formats; and HTTPS to connect it all together. I'll explain each of these building blocks as they come up in the following chapters. The social web isn't separate from the World Wide Web; it's deeply integrated.

A Brief History of the Fediverse

The standards for the social web did not arise out of a vacuum. They came from patterns, specifications, and products that have been important since the beginning of the internet. Knowing these precedents can help you understand how and why the ActivityPub standard works the way it does, but this knowledge is not required, so feel free to skip this section if you just want to get to the coding part.

The *federation* pattern dates back to the origin of the internet. I mean, hey, *internetwork* is right there in the name. The internet started as a computer network crossing organizational boundaries between universities, corporations, and government institutions. Each of these organizations maintained a level of authority within its own network but used open standards to maintain the connections with others. This allowed cool applications, like email, to be built on top of the internet.

The World Wide Web was first launched in 1990 by Tim Berners-Lee at the European Organization for Nuclear Research (CERN) in Switzerland. Like other internet technology, it uses a federated structure: each domain has its own, perhaps very large, structure of documents housed there, managed and controlled by the domain's owner. The web also defines a uniform protocol for exchanging documents, HTTP, and a relatively small set of standard document formats for text, images, video, and audio, plus styling instructions and executable code. By depending on this constrained set of formats, the web allows an explosive amount of client and server software, and even machine-to-machine conversations via web APIs.

As the web grew, websites began updating more frequently than users could keep up with. Netscape Corporation, a major browser vendor, developed a standard for updated content on a website. Really Simple Syndication, or RSS, was picked up by the W3C and, separately, the legendary Dave Winer of Userland Software. With the addition of the related Atom feed format later in the 2000s, developers had multiple ways to convey update feeds across the internet.

This was great, because at the same time blogging was exploding as a phenomenon. A *blog* is a sequential website that displays posts in reverse-chronological order (newest first). Winer calls a blog "the unedited voice of a person" and emphasizes its personal expression as well as technical features. As the 2000s progressed, anyone could set up a blog to share their thoughts, experiences, jokes, code, problems, and ideas with the world. Blogs worked really well with RSS feeds; each blog post was an entry in the feed. Instead of loading up each of your friends' blogs every day, you could use specialized software called an *RSS feed reader* to bring them all together in one place.

Social graph software first arose in the 1990s with services like SixDegrees, which let people define their friend and family relationships and do interesting analysis on the graphs. But in the mid-2000s, new social-network sites like Friendster, MySpace, Twitter, and Facebook arose that combined the social graph, the blog authoring

experience, and the feed-reading experience into one service. This proved exceedingly popular; thousands of these services launched between 2005 and 2010.

The social-network services soon branched out to become social-network *platforms*. They built APIs, data feeds, and embedding mechanisms that let developers from other companies embed widgets onto a user's profile page or inject active applications into their home feeds. It was an interesting, fertile time on social networks!

Almost as soon as social-network services brought the large public populations together on one service, private social-network services started setting up barriers between them. A typical usage was for a single company—an *enterprise social network*—to allow employees to post, comment, share, and follow other staff, but not people outside the network. Millions of enterprise social networks were launched.

Meanwhile, back in the standards world, a new federated standard called Jabber (later, Extensible Messaging and Presence Protocol, or XMPP) was developed by Jeremie Miller to connect chat messaging systems. It defined a structure for a client-to-server interface, a server-to-server interface, and a common extensible data structure. With dozens of server and client implementations and large messaging service support, XMPP provided a great model for how a federated software ecosystem could work.

So, that's a lot of threads; let's try to bring them all back together. Numerous people started open source social-network services toward the end of the 2000s, with varying levels of success. I started one called identi.ca, running software that I created (called StatusNet at the time and now called GNU Social). Similar services and projects began, such as Elgg, Diaspora*, OneSocialWeb, tent.io, Appleseed, DiSo, Buddycloud, Friendica, and many others, often with their own social protocols that didn't interoperate.

A series of informal Federated Social Web Summits in the early 2010s brought social web developers together to share patterns and ideas. The developer community started to coalesce around a group of interesting, interrelated technologies: OpenID, OAuth, PubSubHubbub, Activity Streams (an early version, based on Atom), Salmon, and WebFinger. My team at StatusNet assembled these into a protocol stack we called OStatus, although others call it "the DiSo stack," after a federated social web project started by Chris Messina and Steve Ivy. We also used Activity Streams Atom on top of the Atom Publication Protocol (AtomPub) to implement a robust client API. A brief peak for this period in the federated social web was interoperability between StatusNet, Google's social-network product Buzz, Diaspora*, Tumblr, and others.

OStatus had its benefits but also problems. Most notably, it did not have an effective mechanism for privately distributing content. Everything posted on the network was public. Although this was common at the time, borrowed from the public nature of the blogging network, it inhibited uptake of the technology. People were used to having more fine-grained privacy control in commercial social-network services.

In 2013, a more traditional industry group of enterprise and consumer social-network companies formed OpenSocial, a standards organization for social-network platforms. Although OpenSocial did not define a federation protocol, it did important foundational work in defining standard APIs, so an application or widget developed for one social-network platform could be easily ported to another.

The early 2010s were a nuclear winter for social-network standards. Although some important work was happening, Metcalfe's law and brutal competition winnowed down the massive number of consumer social networks launched in the late 2000s to a few dozen by the middle of the 2010s. As enterprise chat systems like Slack grew, the enterprise social-network market also dried up. With fewer independent social networks, and a few dominant networks with close to a billion users apiece, social-network interoperability seemed like an unnecessary evolutionary dead end.

In this harsh environment, my company StatusNet closed. I hunkered down and wrote a new social software platform called pump.io, including a new federation protocol, using research funds from the Canadian and Quebec government. Unlike OStatus, pump.io used a unified model across all parts of the stack, which made it much easier to work with. In particular, it used Activity Streams' new JSON format, which made it much more like the popular APIs of the time. The seams joined more easily.

In 2014, the OpenSocial organization folded and turned over all its specifications and other assets to the W3C. Spurred on by this development, the W3C chartered a working group to build a stack of social-network standards: a social data standard, a social API, and optionally a social-network federation protocol.

I cochaired the working group with Tantek Çelik and Arnaud Le Hors. The *Social Web Working Group* (*SocialWG*) worked for a grueling three years and eventually published a number of social-network standards—not all compatible. The stack that inspired this book was based on a revision of *Activity Streams,* which was edited by James Snell and me and published in mid-2017. The API and protocol were based on Erin Shepherd's adaptation of the pump.io API and protocol to AS2. Called *ActivityPub*, the spec was edited by Christine Webber-Lemmer and Jessica Tallon, was further refined by Amy Guy, and published in early 2018.

In 2016, meanwhile, Eugen "Gargron" Rochko launched the Mastodon project. A federated open source social-network server, Mastodon originally connected via the good old OStatus protocol to instances of GNU Social and others. When ActivityPub launched, though, the Mastodon team pivoted and adopted the protocol, giving it a first key set of users.

A Tour of the Fediverse Today

Whew! Now that that's over, here we are in the present. As of mid-2024, somewhere around 20 million people have accounts on services on the fediverse. Mastodon remains the most popular software, with some of the most advanced social features, and leads the way in extending the ActivityPub protocol for new uses. A dedicated nonprofit headed by Rochko directs donations into software development and network operations for the big social network, mastodon.social, run by the service. In a big way, the social web depends a lot on interoperability with Mastodon.

Meanwhile, an explosion of social networks has joined the service—around 25,000 services as of this writing. Some are run by companies for their employees, and others are run by individuals for their family and friends, or just for themselves. But many, seemingly the vast majority, are run by individuals or small groups as a public service for general users. Often these servers bring together groups by social or professional affinities—geographically associated services, professional services, or services for women or for LGBTQ+ communities. Some are incorporated as nonprofits or even as member-owned cooperatives. My home service, cosocial.ca, is a member-owned cooperative for Canadians.

Other server software competes with Mastodon for hosting social services. Friendica remains popular, and Pleroma is a great service with low resource requirements. Bonfire, Wildebeest, Takahē, and Misskey provide microblogging-like interfaces with various alternate implementation decisions.

But the types of services available on the fediverse vary widely too. Pixelfed, for example, is a social service focused on photo sharing. PeerTube provides a social video platform. Lemmy and Kbin provide social news interfaces, and BookWyrm is a social book review service. WriteFreely, on the other hand, lets you write long, blog-length text—much more expressive than the limited microblogging on other platforms.

Meanwhile, we see adaptation of more traditional "social" software to the social web. WordPress, the blogging software that powers more than 40% of the domains on the web, has an ActivityPub plug-in that lets WordPress users publish into the social web. Discourse, a forum platform, also lets users link into the fediverse.

With so many communities and so many software platforms, the fediverse has a Wild West vibe to its culture (although it's hard to generalize a community of tens of millions of people). Many of the people on the network have sought a place to establish their own values and to exercise a level of autonomy they can't on other networks. LGBTQ+ people, open source and open standards enthusiasts, and general technology fans seem to find themselves comfortable on the network.

But from my account on cosocial.ca, I follow feeds from the British Broadcasting Corporation (BBC) (on its own dedicated service) and politicians in the United States and Canada. Web information aggregators like Techmeme publish into the network, plus entertainers like George Takei and Morgan Fairchild.

The pace of change over the last few years has been intense. Changes of ownership at commercial social networks, with consequent changes of policy, have put many communities adrift, trying to find a new home. Some have landed on the fediverse. Other incentives have been changes to antitrust and privacy regulations in Europe and the US, which make island-like social-network services less and less tenable.

The influx of new users hasn't always been without controversy. The number of people with accounts on the network has jumped almost tenfold during this time, and many of the old-timers who have been around since 2016 feel like their turf has been invaded. Will the unique culture they've created be spoiled by incoming crowds and newly connected services? Or will the higher level of control that federation brings them let them maintain their own culture with their own rules? Only time will tell.

What the Fediverse Holds in Store for Tomorrow

As I write this, the fediverse is in a time of rapid growth, and we are poised for even more. Announcements from the Mozilla Corporation in December 2022 and Meta in July 2023 that they will be bringing new networks to the social web mean, potentially, another 100 million people or more joining the network, almost overnight. With that many users, and big social players, the fediverse will be attracting more services and more client developers. Another period of tenfold growth is not an unreasonable prediction.

As this next period of the fediverse begins, it's impossible to say exactly what will happen. One possibility is that Mastodon, the current 600-pound gorilla of the fediverse, will cede its status as the dominant platform on the network. Having more participants with equal levels of power might require more collaboration among developers.

Another possibility is that other social-network services, with new gimmicks and parameters, will join the network. They'll bring their own sets of expected behaviors, which might require extensions to the ActivityPub and AS2 standards.

Other services will likely also join the fediverse—or, rather, provide new kinds of interactions that aren't available yet. *Network-wide search*, for example, has proved an elusive goal on a social web, where many users value their privacy more highly than convenience. *Onboarding services* that help find your contacts spread across the fediverse can provide more help for new users. And *social analytics*, services that help you understand your own social graph, can grow as big business, influencers, and other brands will want to measure their success on the fediverse.

Potentially, expansion could reach domains that we don't normally think of as social software. For example, many game consoles include a social platform where players can form connections. Each user has a feed of their activities on the platform—games they've played, scores they've reached, and in-game achievements they've attained—which is shared to each connected player. Could these game networks connect into the fediverse, letting users on other social services follow play? Could they even federate across console platforms?

Another domain is the *Internet of Things* (*IoT*), the network of connected smart appliances found in homes and businesses. These smart devices with embedded microcontrollers can report sensor readings to home hubs or mobile phone apps, but they're hampered by a lack of appropriate standards with broad enough usage. Could this broadcast mechanism, with custom AS2 activities representing sensor readings or device controls, work over the social web?

Once we start talking about devices reporting to people and responding to commands, we can go into areas like enterprise applications or even process control. Can manufacturing systems report the number of products they've created in the last hour to whomever needs to hear it? Could you follow a package on the social network as it moves across the country? Loosely coupling the emitters of data to the consumers of that data, through a publish/subscribe network, is exactly how ActivityPub works. And it might be just what the doctor ordered for enterprise integration.

This is all just speculation on my part, of course. Chapter 6 covers future directions for the fediverse in more depth.

Honestly, social networking is a big enough domain to take on. I love social-network software, and I love the absolutely broad universe of features and apps that software developers have made to connect people. When it comes down to it, our connections to the people we love, our friends, our classmates, our family, our colleagues, and our neighbors are the most meaningful things in life. Making these relationships better, richer, stronger, and more rewarding is the most important use of computer software, by far.

What does the future of the fediverse hold? That's not really up to me, author of this book and dabbler in software and standards. Systems like the social web become truly amazing when developers build software that the system creators never imagined. I hope this book can help you bring your ideas for new fediverse software to fruition. I can't wait to see what you build.

Conclusion

As you can see, ActivityPub has deep roots. However, there's no better time than right now to learn the protocol and understand the ecosystem. The basic architecture—a data format, API, and federation protocol—is easy to understand at a high level. In the following chapters, we'll do a deep dive into each one so that you can be part of creating the future of the social web.

CHAPTER 2

Activity Streams 2.0

Activity Streams 2.0 (AS2) is an essential ingredient of programming for the social web; it is the common data format that all social web applications and services use when communicating with one another. It defines structures that represent common web content types and the activities we perform on them.

This chapter covers how AS2 is structured, what types of data it can represent, and how the standard properties of those data types describe digital and real-world objects.

The First Steps

AS2 is based on JavaScript Object Notation (JSON), pronounced *JAY-son* or *jay-SAWN*, depending on who's saying it and whether someone named Jason is in the room with them. JSON is the most popular data-interchange format on the internet as of this writing in mid-2024. Douglas Crockford defined it in 2002 as a subset of the programming language JavaScript, specifically selected to represent tree-like structured objects. JSON became a formal standard with the publication of request for comment (RFC) 8259 in 2017.

If you're not familiar with JSON, here's a crash course in the format. A JSON file, or *document*, usually looks something like this:

```
{
  "propertyA": "value1",
  "propertyB": 3,
  "propertyC": [4, 8, 15, 16, 23, 42],
  "propertyD": {
    "propertyE": "value2"
  }
}
```

The curly braces { and } delimit a JSON *object*—that is, a structure with named key-value pairs. Each *property* has a name, like `propertyA`, which has to be in double quotation marks. The property also has a *value*, which can be of various types. In the preceding code snippet, I've shown, in order, a quote-delimited string value, an integer value, an array value surrounded by square brackets, and an object value (which itself has a single string-valued property). The name and value of a property are separated by a colon (`:`), and properties are delimited by a comma (`,`). Whitespace characters (like spaces, tabs, and returns) outside of strings are insignificant and can be left out or added to provide more readability.

You can represent all kinds of interesting things in JSON. Using the property-value pairs, possibly with nested objects as values, you can create quite complex and expressive representations of objects. Here's an example of a dog, including its vital statistics and lineage:

```
{
  "name": "Bowser",
  "age": 4,
  "breed": "Labrador retriever",
  "height": 0.6,
  "weight": 30,
  "sire": {
    "name": "Fido",
    "age": 9
  },
  "dam": {
    "name": "Clarissa",
    "age": 8
  }
}
```

There are a few other important things to learn about JSON, and if you're interested, you would definitely benefit from reading Lindsay Bassett's *Introduction to JavaScript Object Notation* (O'Reilly Media, 2015). It's straightforward and covers the topic well.

Most actively used programming languages, as of this writing, have either built-in functions for handling JSON or a standard library for it. In JavaScript, you can usually just use the `JSON` module to work with JSON documents as strings. Here's an example from Node.js:

```
const myJsonString = '{"foo": "bar", "baz": 42}' # a string for JSON source
const myObject = JSON.parse(myJsonString) # string -> Object
const newString = JSON.stringify({"a": 1, "b": 2}) # Object -> string
```

The `JSON.parse` function parses a JavaScript string in JSON format into a JavaScript `Object`. The `JSON.stringify` function goes in reverse, turning a JavaScript `Object` into a JSON string.

In Python, you can use the `json` module to turn JSON-formatted strings into `dicts` and back again:

```
import json
my_json_string = '{"foo": "bar", "baz": 42}'
my_dict = json.loads(my_json_string) # string -> dict
new_string = json.dumps({"a": 1, "b": 2}) # dict -> string
```

JSON is popular for good reason. It's a simple data format without a lot of overhead. It covers a lot of what you need when exchanging data between systems.

Publishers and Consumers

The AS2 ecosystem has two major roles: publishers and consumers. These don't have their regular English meanings (a company that makes books, a person who buys things for personal use), so I want to clarify how the terms are used in this chapter and later.

A *publisher* is any software, program, or company that makes AS2 representations of objects and shares them with others. The publisher is responsible for the format and content of the AS2 object. Publishers can passively make AS2 objects available—say, through a GET endpoint for a REST API. They can actively push AS2 objects to others—for example, through a POST request to a REST API. Nice symmetry there!

Consumers receive AS2 documents from publishers, parse them, and try to make sense of them. That can happen by reading the object from a database, or fetching it from an API, or providing a POST endpoint for an API.

Note that I am intentionally conflating the person or organization responsible for writing the software and the software that's actually doing the work. It's just easier to say that this combined person-software entity has responsibilities and intentions. This is pretty common for people talking about AS2, including in formal specifications, so I am not afraid of introducing you to the concept here.

Publishers and consumers have different responsibilities. Publishers should try their best to make clear, unambiguous AS2 documents that can be properly processed by the most consumers possible. Consumers, on the other hand, should try their best to handle as many different kinds of AS2 documents as possible.

This principle is sometimes stated as, "Be conservative in what you send; be liberal in what you accept." It's called Postel's law, after legendary internet pioneer Jon Postel, who proposed it as a heuristic for all internet software. If publishers make an extra effort to be understood, and consumers make an extra effort to understand others, the chances of successful intercommunication go way up.

Type

We use computers to store and transfer information about all kinds of things: tractors, hockey players, historical events, cans of maple syrup, chairs, astrological signs. For any particular object, we call the kind of thing it is its *data type*, or just *type* for short. The type of the object limits the properties that the object has: you wouldn't expect a can of maple syrup to have a `numberOfLegs` property, but that would make a lot of sense for a chair. A tractor wouldn't have a `jerseyColor` property, but a hockey player would.

An object's type also defines how it can be used; you can't drive an astrological sign, but you could drive a tractor. Objects with one type might be used for one purpose in a computer system, and objects of a different type are used for a different purpose. Data types can get complicated, but as long as you remember that an object's type defines the properties it can have and what can be done with it, you're going to be fine.

One detail plain-vanilla JSON *doesn't* include is any sort of type information for anything besides the standard scalar types (numbers, strings, and so on) and complex types (objects and arrays).

Typically, with REST APIs, we use the context of an HTTP request to know the type of data being transferred. If you use HTTP to `GET https://sales.example/person/1234`, you will probably get back a JSON payload with properties for a person, specifically in a sales context. If you `PUT` to `/person/1234`, your request will probably have a similar JSON payload for a person.

You know this because you signed up for an API key from *sales.example*, and you have a particular goal in building your client application (cha-ching!). The documentation for the API you're using will tell you what type of data, with which properties, you can send or receive from an endpoint.

But AS2 is a standard social data format; it's not specified to come from, or be sent to, any particular API endpoint on any particular service. It can be used in many situations—stored in files, kept in a database, or distributed across a network. We can't depend on context alone to identify the type of data in an AS2 object.

Instead, you use a standard property across all AS2 objects, `type`. This property can have a string value indicating the type of the object:

```
{
  "type": "Person"
}
```

AS2 has more than 50 predefined types of objects, each with its own set of standard properties. It also has more than 60 defined properties; some are common to all types of objects, and some are specific to only one type. The full set of types and properties

makes up the *Activity Vocabulary*, which is covered in detail in Appendix A. You can also define your own types for ActivityPub; I cover that in Chapter 5.

Identity

Here's an AS2 object representing a person named Maria Garcia:

```
{
  "type": "Person",
  "name": "Maria Garcia"
}
```

The `name` property is a standard property in AS2; it is the name or title of the object. It's used for people, videos, software, practically anything. In this case, it's Maria's personal name.

If you've been working in software for a while, you know that a name is not unique enough to unambiguously identify a person. As of this writing, more than 30,000 Maria Garcias live in the US alone. Using the preceding AS2 object as a representation of a person wouldn't be enough to show exactly *which* one it represents.

We could do some other things, like including Maria's photo or other personal information. But even then, making sure the person is unique is hard. In many software systems, records about people have arbitrary unique identifiers so they won't be confused with other people. A Social Security number, passport number, or driver's license number is a common way to specify exactly who the person is.

AS2 uses the `id` property for unique identifiers. The value can be any uniform resource identifier (URI). URIs are a system of identifiers that can handle multiple protocols or naming schemes. You can use any type of URI for this property.

One common URI type is a universally unique identifier (UUID), a string of 36 somewhat randomly generated characters that are unlikely to ever be duplicated:

```
{
  "type": "Person",
  "id": "urn:uuid:842AE42B-3613-4863-91A4-2DA77BFA7B28",
  "name": "Maria Garcia"
}
```

However, in the ActivityPub world, we primarily use HTTPS URLs as IDs. (Every URL is also a URI. Yes, it's complicated!) Here's a more ActivityPub-ish version of Maria's AS2 representation:

```
{
  "type": "Person",
  "id": "https://social.example/users/77785",
  "name": "Maria Garcia"
}
```

Using other ID formats is possible if you're using AS2 separate from ActivityPub. But HTTPS URLs give us a lot of advantages. First, the ID doubles as a permanent place to always find the most up-to-date version of the object; we'll talk more about this in Chapter 3.

URLs also provide a standard *namespacing* mechanism for AS2 IDs, using domain names. Maria's ID is distinct from other IDs like *https://photos.example/users/77785* and *https://blogs.example/users/77785* because the domain names are different. Domain names are an essential part of how we structure the internet, and the openness of the domain name registration system means practically anyone can register a domain name to use for their URLs.

For a given hostname, like *social.example*, the person or organization that runs that service gets to pick the structure of the IDs (or, more likely, the ActivityPub software they use has a specific mechanism for it). You might guess that Maria is user 77785 on the service, based on your previous experience decoding URLs, but that's explicitly none of your beeswax. ActivityPub URLs are opaque; you can compare them against one another as a whole, but you can't break them into their component parts and guess what they mean. This opacity gives the domain owner a lot of freedom in choosing how to lay out the paths for AS2 objects on their server.

As an aside, I want to highlight the format of the URLs I use in this book. For example, *social.example* is not an active domain name for a real website. It's part of a small group of domain names that are used only for examples in documentation. The Internet Engineering Task Force (IETF) RFC 2606 reserves these domains so that no one can put a real web server or any other kind of server behind it. Any domain name with the top-level domain *.example* is reserved, as well as *example.com*, *example.net,* and *example.org*. I try to use only reserved example domain names in this book, but make no mistake, your AS2 object IDs should be real URLs with real domains!

The `id` property lets us define which objects are the same and which are different. Here's an object representing a photo taken by Maria:

```
{
  "type": "Image",
  "attributedTo": {
    "id": "https://social.example/users/77785"
  }
}
```

The `Image` type is for images, like photographs. The `attributedTo` property indicates who made something or is responsible for it.

Vocabulary

Of course, there's no guarantee that if you get a JSON object with a `type` or `id` property, it's going to be an AS2 object. Those are common property names used for all kinds of object-oriented systems; they're not exclusive to AS2 by a long shot.

A *vocabulary* is a collection of types and properties that apply to a particular area of interest. A vocabulary for cars might have types like `Car`, `Driver`, `Part`, and `Manufacturer`, and properties like `paintColor`, `licenseNumber`, `weight`, and `trademark`. AS2 uses a vocabulary specifically for social networking, and it has types for people, software, content objects, and social activities, as well as properties matching each of these types.

To clarify that an AS2 object actually *is* an AS2 object, we use a special property that defines the vocabulary for AS2:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Person",
  "id": "https://social.example/users/evan"
}
```

The `@context` property defines the vocabulary used in the JSON document. That URL, which is the value for the property, is unique to AS2. If you open it in a browser, you'll see a description of all types and properties—known collectively as *terms*—for AS2 objects.

This system of special properties comes from a standard built on top of JSON called JSON-LD. (The *LD* stands for *Linked Data*, which is the modern term for what used to be called the Resource Description Framework, RDF.) There are other ways of defining vocabularies in JSON, but JSON-LD is the one that's most widely used at the W3C, so it's the one we chose when creating AS2.

JSON-LD uses `@type` and `@id` as synonyms for `type` and `id`, respectively. So you could also write the preceding object like this:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "@type": "Person",
  "@id": "https://social.example/users/evan"
}
```

If you're parsing AS2 objects, you should be aware of these synonyms for `type` and `id` and make sure to treat them equivalently. If you're creating AS2 objects, however, you should avoid them; not every AS2 developer is going to remember to parse them correctly, so stick with `id` or `type` to maximize interoperability.

Properties

So, what are some properties of a `Person` object in AS2? I'm going to introduce a few of them at a time throughout this chapter. I've already introduced `name`:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Person",
  "id": "https://social.example/users/evan",
  "name": "Evan Prodromou"
}
```

The `name` property is important for all kinds of objects; you can think of it as a title too.

Another important property for a person is `icon`, a small, square image that represents the person:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Person",
  "id": "https://social.example/users/evan",
  "name": "Evan Prodromou",
  "icon": "https://social.example/users/evan.png"
}
```

You can use just a plain URL as the property value here. But using a whole JSON object to describe the icon instead can be useful:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Person",
  "id": "https://social.example/users/evan",
  "name": "Evan Prodromou",
  "icon": {
    "type": "Link",
    "href": "https://social.example/users/evan.png",
    "mediaType": "image/png",
    "width": 512,
    "height": 512
  }
}
```

Notice a few important things here. First, the icon object has a `type` of `Link`; this is the type used for links to files. You can see the file's URL and some metadata about the image—its `width`, `height`, and `mediaType` (that is, the type of file). This could help a consumer, like a web app, size its viewport for the image accordingly.

Property values in AS2 can be simple and have one main type. For instance, `width` is always a number, and `name` is always a string. But they can be more complex, like `icon`.

Usually, you can count on these properties having one of three main styles: a URL, an object, or an array. For `icon`, for example, an array value might give a consumer icons of different file types, or sizes, to choose from:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Person",
  "id": "https://social.example/users/evan",
  "name": "Evan Prodromou",
  "icon": [
    {
      "type": "Link",
      "href": "https://social.example/users/evan.png",
      "mediaType": "image/png",
      "width": 512,
      "height": 512
    },
    {
      "type": "Link",
      "href": "https://social.example/users/evan-large.png",
      "mediaType": "image/png",
      "width": 1024,
      "height": 1024
    }
  ]
}
```

If you are writing software that will consume AS2, make sure it can handle all three types. For example, the following client-side JavaScript snippet parses an AS2 string and sets the values of an `<img>` element to the icon:

```
function setIcon(str, element) {
  const person = JSON.parse(str);
  switch (typeof person.icon) {
    case "string":
      element.src = person.icon;
      break;
    case "object":
      let icon = Array.isArray(person.icon)
        ? person.icon.sort((x) => Math.abs(x.width - 512))[0]
        : person.icon;
      element.src = icon.href;
      element.width = icon.width;
      element.height = icon.height;
      break;
    default: // undefined, null, a number…?
      element.src = "/assets/default.png"; // set a default icon
      break;
  }
}
```

This process can be frustrating at times for developers new to AS2. Why do you have to write code for three or more alternatives? Why not just have one type for `icon` and make it required, to boot? However, this kind of resilience lets us build more flexible, adaptable code: choosing the right image for a mobile interface versus desktop, say, or for low-memory embedded devices. The publisher can create the representation that closely matches its own internal representation, and the consumer can adapt that representation to its own needs. The expressiveness of AS2 lets everyone get what they need out of the file format.

Activity Types

The most important group of data types in the AS2 vocabulary are activities. *Activities* are like sentences: they say that *somebody* did *something* to *something*.

Here's an example of an activity in Activity Streams format:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/activity/1",
  "actor": {
    "type": "Person",
    "name": "Harriet",
    "id": "https://social.example/person/harriet"
  },
  "type": "Like",
  "object": {
    "type": "Image",
    "name": "Photo of Carol by the campfire",
    "id": "https://social.example/photos/338",
    "url": "https://social.example/uploads/1287.jpg"
  },
  "summary": "Harriet liked the image 'Photo of Carol by the campfire'"
}
```

A lot is going on there, right? But we can break it down by looking at the top-level properties of the activity. We know the `@context` just marks it as an AS2 object. The ID is a unique identifier. Good so far!

The `actor` property is new to us; it's who did the activity. In this case, the property value is a JSON object, an AS2 `Person` type with the name `Harriet`.

The `type` for the activity is what the actor did. In this case, it's the `Like` activity. This is liking something in the social-network sense of the term (sometimes called *faving* or *favoriting*). It's a way for Harriet to give positive feedback to the object's creator.

Then there's the `object` property. That's the thing that Harriet liked: the direct object of the sentence. In this case, it's an `Image` object, with a title `Photo of Carol by the campfire`, an ID, and a URL.

The last property listed is `summary`. We use `summary` to describe AS2 objects that don't have a `name`, although it's OK to have both. The summary gives a description of the activity, in this case, which can be used to display the activity in a text interface.

Almost all activity types follow this format: the `actor` is the subject of the sentence, the `type` is the verb, and the `object` is the direct object.

Activities are the heart of AS2, which is why they're highlighted so prominently in the name. The emphasis on activities shifts the focus of this vocabulary from inert *stuff* like images and text to the lively interactions that make social networks so important and enjoyable. AS2 describes a world in motion, interacting, forming, and breaking connections.

The structure of activities in AS2 is based on activity theory, a psychological framework popular with design theorists. It's quite complex, but one way to summarize this theory is that activities are rich with context—the tools used to do them, the actor's reasons for doing them, the results, who they were for, what was changed. Other AS2 properties for activities describe this context—how the activity happened, indirect objects, and the results. All the activity properties are in the reference in Appendix B.

Actor Types

Another subset of types in the Activity Vocabulary is called *actor types*, because they represent the kind of things that can initiate actions in a social-network setting. You've already seen the `Person` type, which represents any person. Especially in the ActivityPub sense, a `Person` is usually a single user account:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://home.example/person/anneke",
  "type": "Person",
  "name": "Anneke Cherrier",
  "summary": "I am a photographer and designer in the Netherlands.",
  "icon": {
    "type": "Link",
    "href": "https://home.social/uploads/anneke-headshot-2023.jpg",
    "mediaType": "image/jpeg"
  }
}
```

Two other important actor types are `Group` and `Organization`, which represent collections of people. A `Group` is any group of people, but especially an online, informal group, like an interest group:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://home.example/group/evmemes",
  "type": "Group",
```

```
  "name": "Electric Vehicle Memes for Battery-Charged Teens",
  "summary": "News and humor about electric vehicles",
  "icon": {
    "type": "Link",
    "href": "https://home.example/uploads/ev-memes-logo.jpg",
    "mediaType": "image/jpeg"
  }
}
```

An `Organization` is more formal; it's a company, government agency, nonprofit, or other entity with a real-world presence:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://oreilly.example/",
  "type": "Organization",
  "name": "O'Reilly Media",
  "summary": "Gain knowledge and hone skills with O'Reilly",
  "icon": {
    "type": "Link",
    "href": "https://oreilly.example/uploads/company-logo.jpg",
    "mediaType": "image/jpeg"
  }
}
```

Finally, two types represent software. An `Application` is any kind of software, but most often it applies specifically to frontend software, especially software built to use the ActivityPub API described in Chapter 3:

```
{
 "@context": "https://www.w3.org/ns/activitystreams",
 "id": "https://evanp.github.io/ap/client.jsonld",
 "type": "Application",
 "name": "ap",
 "summary": "ap command-line ActivityPub client"
}
```

A `Service` is a service of any kind, such as a delivery service or an ecological service. However, especially for the ActivityPub world, an API server that implements the API described in Chapter 3, or a federated server that implements the protocol in Chapter 4, would be treated as a `Service`:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/",
  "type": "Service",
  "name": "Social Dot Example",
  "summary": "An example social service used in this book"
}
```

As you can probably tell, types in AS2 are much less strict than in strongly typed programming languages. Does `Person` refer to a real person or a fictional person? A person in the legal sense or a person in the colloquial sense? The answer is, it depends.

Usually, these types are applied in social-network situations, making it tempting to think of `Person` as a user, `Organization` as a brand, `Group` as a, well, a group, and so on. These are perfectly reasonable interpretations, and they're widely implemented in the social web. But it's OK to keep your mind open to other uses too.

Activity objects have an `actor` property. The objects referred to by this property usually have an actor type, but it's not strictly required.

Object Types

Object types in the Activity Vocabulary are types related to real-world and digital objects. We've seen the `Image` type:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/avatar/1",
  "type": "Image",
  "name": "Avatar for evanp",
  "url": "https://social.example/upload/evanp1.jpg"
}
```

Note that the `id` of the object is the URL of the JSON representation. The `url` is the full binary version. Yes, they're both URLs, but only one is the `url`! Don't worry, you'll get used to it after a while.

This `url` property value doesn't give very much information about the binary data found there. We can also use a `Link` object for the property value:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/avatar/1",
  "type": "Image",
  "name": "Avatar for evanp",
  "url": {
    "type": "Link",
    "href": "https://social.example/upload/evanp1.jpg",
    "mediaType": "image/jpeg"
  }
}
```

This `url` property value can also be an array of `Link` objects. We might do this to give consumer software a chance to choose image sizes or media types that match its needs:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/avatar/1",
  "type": "Image",
  "name": "Avatar for evanp",
  "url": [
    {
      "type": "Link",
      "href": "https://social.example/upload/evanp1.jpg",
      "mediaType": "image/jpeg"
    },
    {
      "type": "Link",
      "href": "https://social.example/upload/evanp1.png",
      "mediaType": "image/png"
    }
  ]
}
```

Besides `Image`, other similar types are used for digital content. We have a type for `Video`:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/files/12",
  "type": "Video",
  "summary": "Geese flying overhead by the St. François river",
  "duration": "PT23S",
  "url": {
    "href": "https://social.example/upload/geese12.mp4",
    "mediaType": "video/mpeg4",
    "type": "Link"
  },
  "attributedTo": {
    "type": "Person",
    "name": "Evan Prodromou",
    "id": "https://social.example/users/evanp"
  }
}
```

Note the `attributedTo` property of the video. This points to the actor responsible for creating the object—in this case, a `Person`, although it doesn't have to be.

AS2 also has a type for `Audio`:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://music.example/beatles/let-it-be/across-the-universe",
  "type": "Audio",
  "name": "Across the Universe",
  "duration": "PT3M48S",
  "url": "https://music.example/files/01/03/A4/001-87B-0.mp3"
}
```

For text content, AS2 has two main types: `Note` and `Article`. `Note` is interesting and important; it represents a short text of just a few sentences to a paragraph and is commonly used for microblogging or for comments on other content:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/note/331",
  "type": "Note",
  "content": "The Note type can be contained in its own AS2."
}
```

`Note` objects often don't have a URL pointing to more content. All the content is in the AS2 representations, in the `content` property. Longer text is represented as an `Article`:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://blog.example/api/post/331",
  "type": "Article",
  "name": "What I Did Last Summer",
  "summary": "Summer has come and gone. Here's what I did.",
  "url": "https://blog.example/posts/what-i-did-last-summer"
}
```

There's no hard-and-fast limit defining what makes something short enough to be a `Note` or long enough to be an `Article`. The Activity Vocabulary specification says that a `Note` should be less than a paragraph and an `Article` should be multiple paragraphs, but many implementations will allow a small number of paragraphs in a `Note`.

Object types can also represent real-world objects. The `Place` type represents a place in the world, like a city, store, country, or lake:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://osm.example/relation/1634158",
  "type": "Place",
  "name": "Montreal, Quebec",
  "latitude": 45.508888,
  "longitude": -73.561668,
  "altitude": 6.0
}
```

In social software, places are mostly important for giving context to content.

An `Event` is a thing that has happened or is going to happen. Usually, this type is used for planned events, like a party, meeting, or meetup:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://invitation.example/evanp/event/338",
  "type": "Event",
  "name": "Evan's Birthday Party"
}
```

And a `Relationship` is a relationship between people:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://connection.example/evanp/frank",
  "type": "Relationship",
  "subject": {
    "type": "Person",
    "name": "Evan"
  },
  "relationship": "http://purl.org/vocab/relationship/closeFriendOf",
  "object": {
    "type": "Person",
    "name": "Frank"
  }
}
```

Here, `subject` and `object` represent two people. The `relationship` property, which is not the same as the `Relationship` type (note the capitalization!), defines the nature of the relationship. AS2 doesn't have its own vocabulary for relationships, so in this case, I've borrowed the term `closeFriendOf` from the "Relationship" vocabulary. We'll get deeper into using other vocabularies later in the chapter.

Activity types have a property named `object`. The value of this property is usually an object type, but it's not strictly required. The difference between actor types and object types in AS2 is mostly informative—they make good candidates for `actor` and `object`, respectively—but there are a *lot* of exceptions.

HTML

HTML, short for *Hypertext Markup Language*, is a rich text format used as the primary document format for the World Wide Web. HTML uses markup syntax called *tags* or *elements* to add formatting, page structure, and links to plain-text documents. If you aren't familiar with HTML, need a refresher, or just want to be reminded of how cool this language is, I recommend *HTML5 Now* (New Riders Publishing, 2010) by Tantek Çelik or *HTML5: Up and Running* (O'Reilly, 2010) by Mark Pilgrim. (The current version of HTML is 5. It was pretty new back in 2010, which is why it's in the title of both books.)

HTML is allowed in two major AS2 properties by default: `summary` and `content`. For `summary`, HTML is allowed only if the object also has a `name`; otherwise, the summary has to substitute as the main representation of the object. But HTML is (almost) always used for `content`.

HTML is a powerful tool that can provide a rich user experience, making readable content that links to the rest of the web. That's a double-edged sword, however. Some

HTML features, such as stylesheets or dynamic content with JavaScript, can interfere with the functionality of the client displaying the HTML.

Before showing the `content` or `summary` of an AS2 object in an HTML widget or a web browser, then, it's a good idea to *sanitize* it first. HTML sanitizers take out most of the complicated tags and attributes that provide dynamic features. Good HTML sanitizer libraries are available for most programming languages; sanitize-html on Node.js or html-sanitizer in Python are good starting choices.

The AS2 specification does not limit the HTML tags and attributes that can be included in ActivityPub `content` or `summary` properties. However, as of this writing, Mastodon allows a subset of the HTML tags and attributes, which I'm going to replicate here since I mostly agree with these choices:

- `<p>`
- `<span>` (`class`)
- `<br>`
- `<a>` (`href`, `rel`, `class`)
- `<del>`
- `<pre>`
- `<code>`
- `<em>`
- `<strong>`
- `<b>`
- `<i>`
- `<u>`
- `<ul>`
- `<ol>` (`start`, `reversed`)
- `<li>` (`value`)
- `<blockquote>`

With this set of tags, you can get a document-like interface with paragraphs, text formatting, numbered and bulleted lists, and even source code examples. That should usually be enough for a social media status update or comment, or even a multiparagraph blog post.

The main takeaway here is that using HTML is cool and fun, but interacting with other people on the social web means we need to keep safe operations as a primary goal.

Attachments and Tags

Content objects like `Note` and `Image` will sometimes have one or two important additional properties that add a relationship between the content object and the object(s) referred to in the property.

The first is `attachment`, which lets users "attach" related content to an object. Here's an example `Note` with two attached `Image` objects:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/note/35",
  "type": "Note",
```

```
  "content": "We went to the lake this weekend. So fun!",
  "attachment": [
    {
      "type": "Image",
      "id": "https://social.example/image/45",
      "summary": "On the sailboat, sun in our eyes",
      "url": {
        "type": "Link",
        "href": "https://social.example/uploads/image45.jpg"
      }
    },
    {
      "type": "Image",
      "id": "https://social.example/image/49",
      "summary": "Back on the beach, eating oranges",
      "url": {
        "type": "Link",
        "href": "https://social.example/uploads/image49.jpg"
      }
    }
  ]
}
```

The relationship here is a lot like attachments in email. They can either be secondary, supporting the main content object, or conversely they can be the whole focus, and the main content object is just there to introduce and organize them.

The other major property that introduces this kind of relationship is `tag`. This is for identifying people, places, or topics related to the content object. The relationship is looser than the attachment—more like a link or reference. Here's a `Video` with two tags:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://movie.example/video/77",
  "type": "Video",
  "summary": "<a href='https://a.example/users/evan'>
             @evan@a.example</a> on
             <a href='https://tags.example/haha'>#haha</a>",
  "tag": [
    {
      "type": "Mention",
      "href": "https://a.example/users/evan",
      "name": "@evan@a.example"
    },
    {
      "type": "Hashtag",
      "href": "https://tags.example/haha",
      "name": "#haha"
    }
  ]
}
```

This video has two `tag` items. The first is a `Mention`, which represents a mention of a person (usually) on the social web. The second is a `Hashtag`, which represents a loose topic (here, hilarity). Both of these types are related to `Link`, so they use `href` instead of `id` for identity.

These two types of tags are common in social networking; Chapter 3 has more information about social microsyntax.

Tags and attachments are a good way to establish loose and tight relationships between objects. They're both common on the social web, so new AS2 consumers should be aware of them.

Collections

As mentioned previously, JSON can represent an array of items: a series of numbers, strings, objects, or even other arrays. Arrays are pretty useful but have some downsides; on their own, they can't have properties, like an ID or a name or description. And if you use an array to store a group of things, you need to include *all* those things in your JSON document; there's no way to say, "This is some of the things, and there are more over there." When the number of items in a group gets very large, into the tens or hundreds of thousands, including all of them in a single document is unwieldy.

To allow identity, metadata, and partial representations, among other benefits, AS2 expands the idea of an array into a *collection*. The `Collection` type can include information about the size and structure of the group of objects. Here's a collection representing all my brothers:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/lists/brothers",
  "type": "Collection",
  "summary": "All of my brothers",
  "totalItems": 3,
  "items": [
    {
      "id": "https://social.example/users/andyp",
      "type": "Person",
      "name": "Andy"
    },
    {
      "id": "https://social.example/users/tedp",
      "type": "Person",
      "name": "Ted"
    },
    {
      "id": "https://social.example/users/natep",
      "type": "Person",
      "name": "Nate"
```

```
    }
  ]
}
```

You're familiar by now with `@context`, `id`, and `type`, and you've seen `summary` before. The important new properties here are `items` and `totalItems`. The `items` property is a JSON array, with one object of type `Person` for each of my brothers. The `totalItems` property gives the number of items in the collection (in this case, `3`).

If a `Collection` object contains many, many items, we can use *paging* to take the items a chunk at a time. A *page* is a subset of the whole collection; it includes its own `items` property, as well as metadata properties to show how to get more items.

Let's take an example collection: all the cities and towns in California. As of this writing, the state has 482. If I tried to include the whole list as an example, it would fill up a big chunk of this book! But I can use a paged `Collection` object to represent them instead:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://places.example/us/ca/cities",
  "type": "Collection",
  "summary": "California cities",
  "totalItems": 482,
  "first": {
    "id": "https://places.example/us/ca/cities/1",
    "type": "CollectionPage",
    "summary": "First page of California cities"
  }
}
```

This is similar to the brothers collection, but there's a key difference: instead of including an `items` property with an array of objects, one for each item, this collection includes a `first` property. It's an object of type `CollectionPage`, which is the first page of the collection.

Here's a longer version of this page:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://places.example/us/ca/cities/1",
  "type": "CollectionPage",
  "summary": "First page of California cities",
  "items": [
    {
      "id": "https://places.example/us/ca/adu",
      "type": "Place",
      "name": "Adelanto"
    },
    {
      "id": "https://places.example/us/ca/agh",
```

```
      "type": "Place",
      "name": "Agoura Hills"
    },
    {
      "id": "https://places.example/us/ca/ngz",
      "type": "Place",
      "name": "Alameda"
    }
  ],
  "next": {
    "id": "https://places.example/us/ca/cities/2",
    "type": "CollectionPage",
    "summary": "Second page of California cities"
  },
  "partOf": {
    "id": "https://places.example/us/ca/cities",
    "type": "Collection",
    "summary": "California cities"
  }
}
```

`CollectionPage` holds the `items` for the `Collection` (some of them, at least).

Let's look at two other important properties of `CollectionPage`. The `next` property shows the next page in the collection; we would expect to see a few more cities from California in there. A normal way to get *all* the items in a paged collection is to load the `first` page, then its `next` page, then *that* page's `next` page, and so on, until you load a page without a `next` property, which must be the end.

`CollectionPage` also includes the `partOf` property, which is the `Collection` this page is a part of. That gives us a chance to navigate "up" as well as "forward," and get more information about the page.

Collections are usually treated as a *sequence*: the order of the items and of the pages matter. If emphasizing the ordered nature of the collection is important, however, you can use a related set of types: `OrderedCollection` and `OrderedCollectionPage`. Here's an ordered collection of Nobel Peace Prize winners:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://nobel.example/prizes/peace/winners",
  "type": "OrderedCollection",
  "summary": "Nobel Peace Prize winners",
  "totalItems": 138,
  "first": {
    "id": "https://nobel.example/prizes/peace/winners/1902",
    "type": "OrderedCollectionPage",
    "summary": "Peace Prize winners of 1902",
    "orderedItems": [
      {
        "id": "https://nobel.example/laureates/dunant",
```

```
        "type": "Person",
        "name": "Henry Dunant",
        "summary": "Founder of Red Cross"
      },
      {
        "id": "https://nobel.example/laureates/passy",
        "type": "Person",
        "name": "Frédéric Passy",
        "summary": "Founder of Inter-Parliamentary Union"
      }
    ],
    "next": "https://nobel.example/prizes/peace/winners/1903"
  }
}
```

Here, the type of the main object is `OrderedCollection`, and the type of its `first` property is `OrderedCollectionPage`. Similarly, the `orderedItems` property is the ordered counterpart of the `items` property.

These data types work almost exactly the same as their unlabeled counterparts, except with an emphasis on the order. In addition, the page includes its items embedded in the representation in the `orderedItems` property. Although this is the first time I've mentioned it, this kind of embedding is actually a common style in AS2. It lets the consumer get a few free items with their first load of the collection.

Using the `Collection` or `OrderedCollection` types is mostly a matter of style. If you're representing a real-world collection that doesn't have any type of order, it's best to use `Collection`. And if you want to be sure that everyone knows your collection is ordered, use `OrderedCollection` and friends. I find that I primarily use `OrderedCollection`, even though the names of the types and properties are kind of wordy. It lets me express the order of my collections without any doubt.

Addressing Properties

Many types of objects are made *for* someone. For example, a `Note` might be directed to just a few people, or a `Video` might be made for the general public. A `Like` activity might be only for the creator of the content being liked ("I wanted to let you know I liked your song") or it might be for the entire world ("I want everyone to know that I like this song!").

Here's an example of a `Note` for a particular person:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evan/note/1",
  "type": "Note",
  "attributedTo": {
    "id": "https://social.example/users/evan",
```

```
    "type": "Person",
    "name": "Evan P"
  },
  "to": {
    "id": "https://social.example/users/maj",
    "type": "Person",
    "name": "Michele J"
  },
  "content": "We are out of maple syrup!"
}
```

The new addition is the `to` property. Its value is an object that represents the person or other type for whom the note is intended. The value of the `to` property doesn't have to be a `Person`; here's an example of an image posted to a `Group`:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evan/image/1",
  "type": "Image",
  "attributedTo": {
    "id": "https://social.example/users/doris",
    "type": "Person",
    "name": "Doris P"
  },
  "to": {
    "id": "https://social.example/groups/park-slope-moms",
    "type": "Group",
    "name": "Park Slope Moms"
  },
  "name": "Have you seen this cat?",
  "url": "https://social.example/uploads/missing-cat-6645.jpg"
}
```

A few related properties are used for addressing objects and activities: `to`, `cc`, `bto`, and `bcc`. Table 2-1 shows roughly how they relate to one another.

Table 2-1. Properties for addressing objects and activities

	Public	Private
Primary	to	bto
Secondary	cc	bcc

The parallels to internet email are intentional; the semantics of these properties are roughly the same as the RFC 5322 mail header names. The `to` property is the most direct; it is the intentional addressee for the object. If you send something to someone, this is the property to use for them.

The `cc` property identifies a secondary addressee; the content or activity is not for them, but they're getting a copy to view for themselves, kind of as an FYI. The name comes, via email, from an old technology for making carbon copies of a document by

putting a messy, fragile piece of carbon paper behind your typing paper when you type a letter; the *cc* addressee got the faint, blurry copy of the letter. Don't worry; `cc` recipients in ActivityPub get nice, clear, perfect copies of whatever is addressed to them.

If you reply to someone's `Note`, for example, the `to` of your reply might be the author of the note, and the `cc` might be anyone *they* have replied to. Sometimes the `cc` is the only addressee; this is common for open posts on the social web, where the public, or one's followers, are allowed to see content you write or make mostly for yourself.

The properties `bto` and `bcc` are the equivalent of `to` and `cc`. However, they're defined to be visible only to the author of an object. They aren't shown to anyone else, not even the actors identified by these properties. (In Chapters 3 and 4, I'll talk about how properties are shown to anybody.)

The addressing properties can, and often do, have multiple values. For example, here's a note sent to two addressees:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evan/note/1",
  "type": "Note",
  "attributedTo": {
    "id": "https://social.example/users/evan",
    "type": "Person",
    "name": "Evan P"
  },
  "to": [
    {
      "id": "https://social.example/users/amita",
      "type": "Person",
      "name": "Amita P"
    },
    {
      "id": "https://social.example/users/stavro",
      "type": "Person",
      "name": "Stavro P"
    }
  ],
  "content": "Did either of you finish the maple syrup?"
}
```

This `to` property consists of two objects, each of which is considered equally primary. The square brackets show the array. ActivityPub has a special address value that represents the general public. The value is defined this way:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://www.w3.org/ns/activitystreams#Public",
  "type": "Collection"
}
```

The value can also be represented with the short name `Public` or with the prefixed name `as:Public`. Of the three, `as:Public` is the most correct, and it's what I'll try to use in this book. But all three forms are common on the social web.

`as:Public` means *everybody*. That doesn't mean that ActivityPub servers try to deliver objects with `as:Public` in the addressing properties to literally everybody; it does mean that when the object is requested, (almost) anyone is allowed to read it.

The addressing properties are an essential part of the authorization framework for ActivityPub, as you'll see in Chapter 3. In a nutshell, the creator of an object can make modifications or delete the object, but the addressees can read and/or react to it.

Don't worry too much about the addressing properties and their fine distinctions. Loading all your addresses into `to` is perfectly OK. But make sure to get those addresses in *somewhere*, or your recipients will never get their notes.

Internationalization

Even at this point in the development of the internet in 2024, many English speakers still focus only on monolingual English-language content. But billions of people are on the internet, only about 5% of whom speak English as a first language. Especially in social interactions, like the ones AS2 supports, people want to converse in their home language. After all, that's what a home language is: the language we use with friends, family, colleagues, neighbors, and other people we care about.

AS2 has a feature that lets you mark text as being in a particular language and provides alternatives in different languages. Let's take, as an example, the `name` property for a `Place` object named `London`:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://places.example/city/london",
  "type": "Place",
  "name": "London"
}
```

Here's how to show that the name for this city is in English:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://places.example/city/london",
  "type": "Place",
  "nameMap": {
    "en": "London"
  }
}
```

In this second example, the `name` property has been replaced with the `nameMap` property. Instead of a string, the `nameMap` property has a JSON object as its value. This object's property names are language codes, and the property values are strings representing the name in that language. That is, the object "maps" the language code to the name in that language. The preceding snippet shows that `London` is the name for this place *in English*, and that it might have other names in other languages.

Why is it important to show that the name is English? It helps the user-interface (UI) software showing this place to understand the best way to display the name, which font to use, whether to optimize for right-to-left or left-to-right layout, and so on. A text-to-speech processor could choose a voice with the correct pronunciation and accent. And backend software that indexes, sorts, or searches for place names would know to use English-language rules for abbreviations, plurals, sort order, and other grammatical changes to the name.

Let's add some of those other names to the `nameMap`:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://places.example/city/london",
  "type": "Place",
  "nameMap": {
    "en": "London",
    "fr": "Londres",
    "zh": "伦敦"
  }
}
```

This snippet states that the name of the `Place` in English (`en`) is `London`, and its name in French (`fr`) is `Londres`. The code also gives the name in Chinese (`zh`). Because JSON uses UTF-8 as a character set, just about any human-language alphabet or symbol set can be used.

Two other important properties have `Map` alternatives. First, the `summary` property can be replaced by a language map, `summaryMap`. Here's an example of a `Like` activity with a `summaryMap` property:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/like/3",
  "type": "Like",
  "to": ["https://social.example/users/sylvain"],
  "cc": ["https://www.w3.org/ns/activitystreams#Public"],
  "actor": "https://social.example/users/evanp",
  "object": "https://social.example/users/sylvain/image/13",
  "summaryMap": {
    "en": "Evan liked Sylvain's photo"
  }
}
```

By giving a little more context on the summary, this formulation automatically lets other processors know how to use the summary provided or whether they will need to provide their own summary based on the user's preferred language.

Providing multiple summaries within the object itself is also possible. In this case, because the actor's primary language is English and the activity is addressed to someone whose primary language is French, we can provide summaries in both English and French:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/like/3",
  "actor": "https://social.example/users/evanp",
  "to": ["https://social.example/users/sylvain"],
  "cc": ["https://www.w3.org/ns/activitystreams#Public"],
  "object": "https://social.example/users/sylvain/image/13",
  "summaryMap": {
    "en": "Evan liked Sylvain's photo",
    "fr": "Evan a donné un mention 'J'aime' au photo de Sylvain"
  }
}
```

It's not necessary (or even tractable) to provide summaries in *every* language for *every* activity. It's usually fine to use just the single-entry version of the `summaryMap` property and let other processors deal with missing language codes as needed.

Finally, the `content` property has a map analogue, `contentMap`, which you can use to tag the language of a `content` property:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/note/3",
  "type": "Note",
  "attributedTo": "https://social.example/users/evanp",
  "to": ["https://www.w3.org/ns/activitystreams#Public"],
  "contentMap": {
    "en": "Hello, World!"
  }
}
```

The preceding snippet makes it clear that the classic greeting contained in the `Note` object is in English. Processors can now sort, pronounce, and display that content in an appropriate way. Here's a multilingual version:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/note/3",
  "type": "Note",
  "attributedTo": "https://social.example/users/evanp",
  "to": ["https://www.w3.org/ns/activitystreams#Public"],
  "contentMap": {
```

```
    "en": "Hello, World!",
    "fr": "Bonjour, monde !"
  }
}
```

Unless the creator makes multiple-language alternatives of the `Note`'s content, providing multiple translations is usually not necessary or even expedient. One exception might be for an addressee whose preferred language is known in the `to` line, `cc` line, or other addressing properties.

A special language code, `und`, is used when the original language of a string is undetermined:

```
{
"@context": "https://www.w3.org/ns/activitystreams",
   "id": "https://social.example/users/evanp/note/3",
   "type": "Note",
   "attributedTo": "https://social.example/users/evanp",
   "to": ["https://www.w3.org/ns/activitystreams#Public"],
   "contentMap": {
      "und": "hocus-pocus"
   }
}
```

The language features of AS2 can be confusing. Here are some good rules of thumb:

- Use `summaryMap` instead of `summary` if you know the language of the summary, especially if the string is automatically generated.
- Use `contentMap` if you know the language of the content.
- Use `name` by default for people. (Despite what I was taught in high-school Spanish class, in most cases, people's names don't need to be translated.)
- Use `name` by default for other types of objects, unless you know that the `name` for the object is different in different languages.
- You will rarely need more than one entry in the language map.

You can get a lot of benefit from using the language-map feature of AS2, as long as you use it in moderation. Keep your translations limited to the audience you're trying to reach.

Timestamps

AS2 has two important timestamp properties. The `published` property defines when the object was created or first distributed digitally:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Image",
```

```
  "summary": "A blue-flag iris in the sun",
  "id": "https://photos.example/photo/17880",
  "published": "2024-06-02T14:16:00Z",
  "url": "https://photos.example/files/17880-137.jpg"
}
```

Notice a couple of points here. First, the published timestamp is in ISO 8601 format. This is a string that combines the date, time, and time zone. The full format is *YYYY-MM-dd*`T`*HH:mm:ss*`Z`:

- *YYYY* is the four-digit year.
- *MM* is the two-digit month.
- *dd* is the two-digit day.
- `T` is a constant separator.
- *HH* is the two-digit hour on a 24-hour clock.
- *mm* is the two-digit minute.
- *ss* is the two-digit seconds.
- `Z` is a constant time-zone format for Coordinated Universal Time (UTC).

Most programming languages have standard libraries to parse these strings into date structures and to convert date structures into ISO 8601 strings.

The other detail to notice is that the `published` timestamp usually indicates when the AS2 representation of the object was first created, not necessarily when the object itself was created. In this case, `published` gives the time when the file was uploaded, not when the photo was taken. For a `Person` object, the timestamp should indicate the time the account was created, not the person's birthdate.

The `updated` property shows when an object was updated. If it has never been changed, the value is either identical to the `published` timestamp or left out altogether:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Note",
  "id": "https://discussion.example/note/339",
  "content": "Anyone seen my hat? UPDATE: Found it, thanks!",
  "published": "2024-06-02T14:29:00Z",
  "updated": "2024-06-02T16:44:00Z"
}
```

A few other properties in the Activity Vocabulary use this date format; you can read more about them in Appendix B.

Type Hierarchy

AS2 has a hierarchy of types. All the objects that have one type are a subset or superset of all the objects that have another type. The smaller, more specific type is called the *subtype*, and the larger, more general type is called the *supertype*. We sometimes say that a subtype *extends* its supertype, since it adds some additional properties or meaning (Figure 2-1).

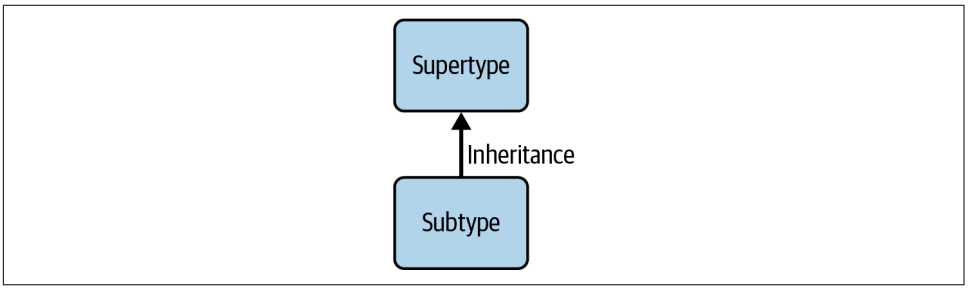

Figure 2-1. A supertype and its subtype

AS2 has two *base* types—types that all other objects are subtypes of. The first one is `Link`, which you've seen previously. It's used for giving basic information about a web resource, based on its URL. Figure 2-2 illustrates the `Link` type and its subtypes.

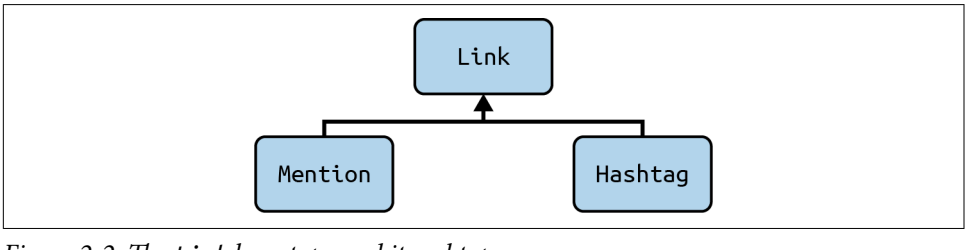

Figure 2-2. The `Link` base type and its subtypes

The second base type is `Object`. Any AS2 object that isn't a `Link` is a subtype of `Object`. For example, `Person`, `Image`, and `Relationship` are all subtypes of `Object`. The `Object` type has more than 70 subtypes, some of which are shown in Figure 2-3.

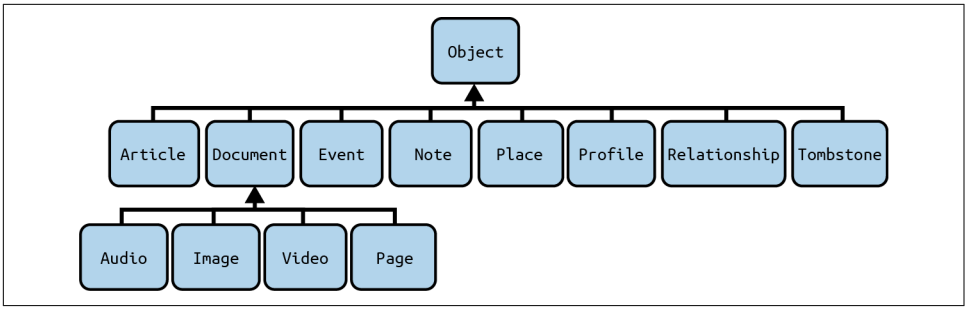

Figure 2-3. The `Object` base type and some of its subtypes

`Collection` is also a subtype of `Object`. It has two subtypes: `OrderedCollection` and `CollectionPage`. Both have the same subtype, `OrderedCollectionPage` (Figure 2-4).

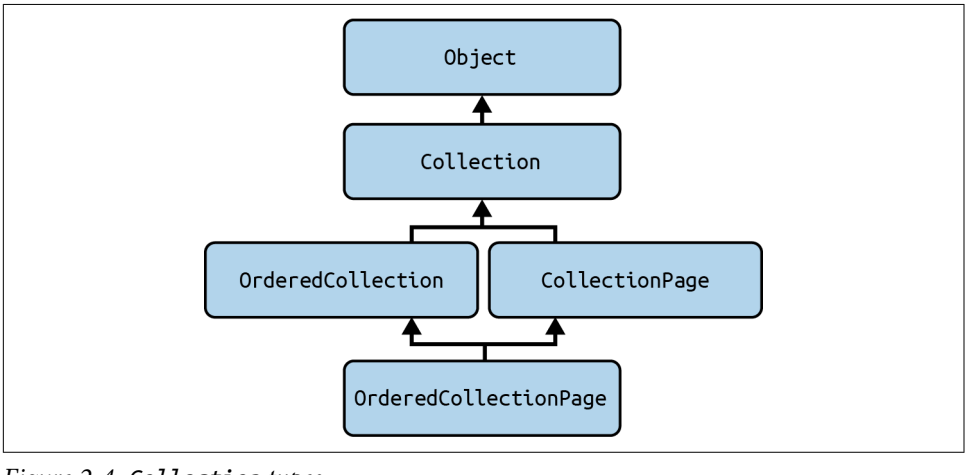

Figure 2-4. `Collection` types

`Activity` is also a subtype of `Object`. Many activity types are a direct subtype of `Activity`. Another abstract type, `IntransitiveActivity`, is used for activities that don't have an `object` property. `Arrive`, for example, is a subtype of `IntransitiveActivity`, which is a subtype of `Activity`, which is a subtype of `Object`. A few other activities are subtypes of more general activities; `Invite` is a subtype of `Offer`, for example. This is pictured in Figures 2-5 and 2-6.

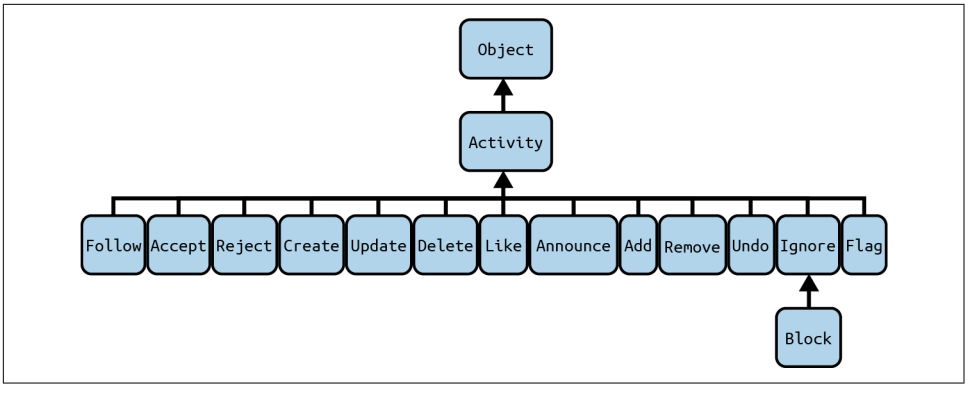

Figure 2-5. Activity types

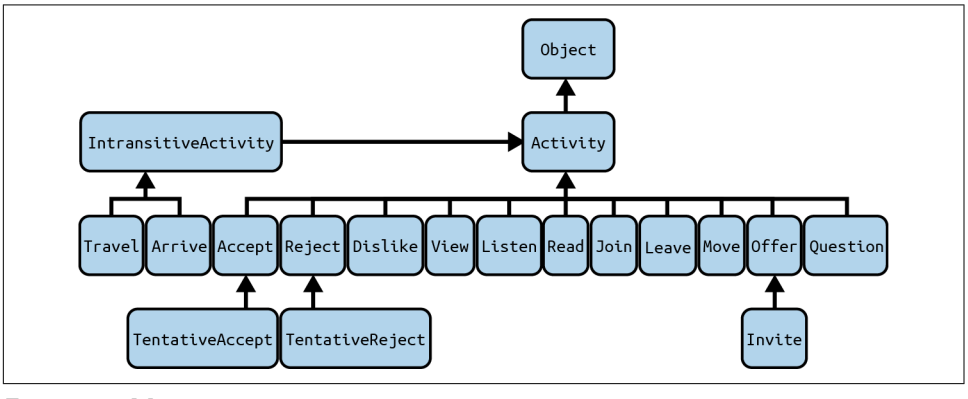

Figure 2-6. More activity types

Finally, as discussed earlier in this chapter, AS2 has actor types, shown in Figure 2-7.

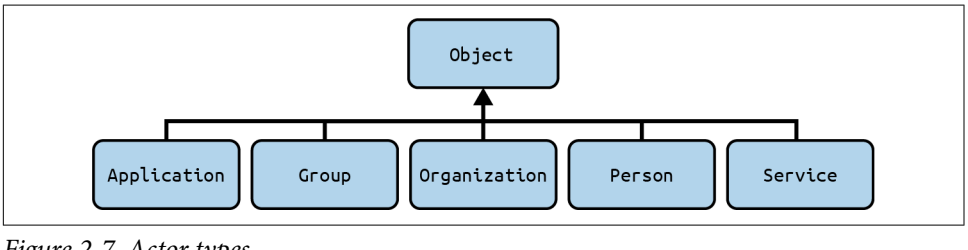

Figure 2-7. Actor types

If you're freaking out right now, please take a deep breath. This hierarchy will *not* be on the test. You don't need to memorize the exact type hierarchy to get value out of AS2's vocabulary.

The main reason that the hierarchy is important is that it lets you know which *properties* apply to which types. Generally, each property has a *domain*, which is a set of types that it applies to. It also has a *range*, which is a set of types that its value can have, but we don't use that as much. Only objects of that type or its subtypes can have that property.

The good news is that, in AS2, most of the interesting properties are waaaaaay high up on the type hierarchy. They're properties of `Object`, `Link`, `Activity`, `Collection`, or `CollectionPage`. Just a few apply elsewhere. This means that, in general, you should check your AS2 objects to make sure that their properties apply to their type. Just know that the answer is often going to be yes.

External Vocabularies

People new to AS2 sometimes feel that its vocabulary is kind of sparse. For instance, for a `Person`, you can add their `name` and `icon`, but what about birth date, phone number, hair color, or height? For those, you'll incorporate properties from *external* vocabularies into your AS2 objects.

In object-oriented design, this technique is called using mixins—useful types that "mix in" with your main types to give them extra properties or behavior. In AS2, we mix in properties from other vocabularies that might be useful.

A *vocabulary*, here, is a collection of types and their properties that models a particular area of interest. The Activity Vocabulary, which we've been discussing in this chapter, models interactions on a social network. Other vocabularies model things like jobs, buildings, or copyright agreements.

The vCard vocabulary, for example, models the closely related area of contact lists or address books. Each `Individual` can have properties that make sense for a contact list, like street address, phone number, job title, and employer.

To include an external vocabulary into an AS2 document, use `@context`. This property is usually a single string—the URL of the AS2 context document. You can also structure it as a list. Here's an example `Person` with the vCard vocabulary included:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "vcard": "http://www.w3.org/2006/vcard/ns#" }
  ],
  "type": ["Person", "vcard:Individual"],
  "id": "https://social.example/users/evanp",
  "name": "Evan Prodromou",
  "vcard:hasAddress": {
    "type": "vcard:Home",
    "vcard:locality": "Montreal",
```

```
      "vcard:region": "Quebec",
      "vcard:country-name": "Canada"
    },
    "location": {
      "id": "https://places.example/canada/quebec/montreal",
      "type": "Place",
      "name": "Montreal, Quebec, Canada"
    }
  }
```

A lot is going on here, so let's break it down step-by-step. The first detail to notice is `@context`. Instead of a single string, it's now an array. You'll recognize the first element too; it's the context document for AS2. The second element is an object with a single property, `vcard`, with a value that's a URL. This object says that this document will use the prefix `vcard` for properties and types from this vocabulary.

Next, you see the `type` property. Instead of the string that you've seen so much of previously, you see an array. The first element is a string that you've seen before for a person. The next one has a prefix value, `vcard:` plus `Individual`, the type most often used for a person in the vCard vocabulary.

Prefixes alleviate conflicts between the two vocabularies. For example, both AS2 and vCard have types called `Group`, which are slightly different in meaning. To indicate that something is a vCard group, use `vcard:Group` to distinguish it. Because we use the AS2 context without a prefix in `@context`, we can use the AS2 version of `Group` without a prefix.

If you want to, you can include the predefined prefix `as:` on any AS2 term you want. That may be helpful in dealing with naming conflicts in some circumstances; on the downside, it may also confuse simple consumer software looking for `Group`, not `as:Group`. The one property that is good to prefix by default is `as:Public`, as mentioned previously.

The multiple values in the `type` property mean that the object defined here is both a `Person` and an `Individual`. You can apply any properties that work for either type. We've "mixed in" the `Individual` type so that we can use its properties in this document.

Next up is `name`, which you've seen before. vCard has a whole cluster of interesting name properties, but generally it's good to use AS2 properties whenever possible. So, here, we have the `name` property from AS2.

The following property has an object as its value. The property uses the `vcard` prefix, and each of its value's properties is a vCard property for defining an address. The object's type is also from vCard, which is `Home` (for home address).

Finally, the `location` property is the AS2 property defining a location related to the object. In this case, the location is a `Place` with an `id` and a `name`. This is equivalent to

the `hasAddress` property, but without the level of detail that the address element has; the city, province, and country are all mixed together in the name. This gives you a good fallback in case a consumer doesn't understand the vCard vocabulary.

The good news is that you can use a lot of interesting vocabularies like vCard for mixing into AS2 objects. The bad news is that, when you publish AS2 objects, consumers will be able to understand and act on only the vocabularies that they know about. So, adding an address to a `Person` with the vCard vocabulary won't automatically add that address to the UI of a web page displaying that person, for example, unless the software creating that page knows about vCard and is prepared to use it.

Chapter 5 goes into detail about using external vocabularies to extend AS2. Here are some simple guidelines for using external vocabularies:

Use them sparingly
: You can add more than one vocabulary to a document, with different prefixes, but using too many can get confusing.

Don't use external vocabulary properties when an AS2 property will do
: As seen previously with the `name` property, it's best to use AS2 terms when possible. This maximizes the chance that AS2 consumers will understand your object.

Use AS2 fallbacks if possible
: If you use properties or types from other vocabularies, try to use related AS2 fallbacks. An example is the `location` property in the preceding example, which is less specific than the `address` property but helps consumers that don't recognize vCard. For types, at the very least, you can use abstract types like `Object`.

Use standard vocabularies
: AS2 consumers are more likely to understand vocabularies that have been through a standardization process. Chapter 5 details some great vocabularies to use with Activity Streams.

Internet Media Types

One magical aspect of the internet is that many kinds of data—images, videos, structured data, and text documents—are transferred through the same protocols. We use the same HTTP message structure to transfer PNG images as we do for MP4 audio. To let software on both ends of the conversation know what's coming through the pipes, we include a *media type* in the protocol. The media type has this format:

(*general type*)/(*specific type*)

A PNG image is labeled `image/png`, and an MP4 file is `audio/mp4`. In certain cases with even more specific refinements to the type, you can see this:

(*general type*)/(*specific type*); *property*=*value*

An example is `text/html; charset=utf-8`, which is the type for HTML text content with the UTF-8 international character set.

For AS2 documents, you can use several internet media types. The basic one is `application/json`. This type is used for any kind of JSON document, AS2 or otherwise. It's a form of *application data*—that is, structured data used for documents. To get more specific, you can use `application/ld+json`. This is the refinement of the JSON media type to JSON-LD. A processor that gets the information that the document is JSON-LD will know how to handle contexts, prefixes, and other JSON-LD idioms in the file.

The JSON-LD type allows even more specific typing by adding a `profile` property. So, an even more refined type for AS2 documents is `application/ld+json; profile="https://www.w3.org/ns/activitystreams"`. This shows that the document is JSON-LD with the AS2 profile.

However, this media type is kind of long and unwieldy. AS2 defines an abbreviation, `application/activity+json`. This is exactly the same as the longer JSON-LD type with the `profile` parameter.

As a consumer receiving AS2 documents, your software will need to be able to handle all these variant types: `application/json`, `application/ld+json`, and `application/activity+json`. As a publisher, you should choose the most specific type that your publishing system will support. If you're not using a preexisting publishing system, `application/activity+json` is usually the best one to use.

Representation Granularity

When publishing AS2 objects, deciding how much detail to include in your representation can be confusing. As you've seen, it's not *always* necessary to include *all* of an object's properties in its representation. Instead, you can include just the right properties, balancing the consumer's need for detail with the system's performance concerns around transferring too much unneeded data.

If the consumer doesn't have the right amount of detail, it will request a more complete representation of the object—maybe from a database server or a web API. That will take valuable time. Giving the representation enough detail could save network requests. But, conversely, if you pack every possible related property into your AS2 object representations, including all the related objects and all *their* properties and related objects, you can transfer way too much data that nobody will ever see or use.

So, here are some rough outlines of different "sizes" of AS2 representations. These aren't official specifications; they're my personal guidelines for finding the right balance between minimal content and fewer requests.

ID-String Representation

The smallest representation of an object is an `id` string. Here's the `actor` property of an activity object:

```
"actor": "https://social.example/users/evanp"
```

This fragment of JSON isn't an object on its own; I'm just focusing on a single property to make a point.

Almost any use at this level of representation is going to require the consumer to make a web request to resolve the full representation of the object (and that will work only if you're using resolvable ID formats, like HTTPS URLs).

You might use this representation if you're storing the activity in a document database, for example, and you don't want to waste space by including any information about the actor, since its full representation might be stored in another document. Or you could use this representation if you know that the consumer already has a fuller representation of the AS2 object, and you just want to let the consumer know *which* object it is. For example, in a collection of 20 activities by the same person, you could use an ID for the `actor` property of each.

You can also use an ID string if you want to force the consumer to fetch data from your server—say, to make sure they always have the most up-to-date version of the object or to make sure they are authorized to access it. This doesn't always work (the consumer can just cache the data it fetches), but it can help in some situations. Again, a trade-off exists between control and performance. Finally, you could use this representation if you don't know anything else about the object and don't want to bother finding out.

Brief Representation

A better minimal representation is what I call a *brief* representation. It has just enough information to let the consumer make some essential decisions about how to handle the object and maybe how to represent it in a text medium. Here's a brief representation of that `actor` property:

```
"actor": {
  "id": "https://social.example/users/evanp",
  "type": "Person",
  "name": "Evan Prodromou"
}
```

This representation has just three properties: an `id` to determine the identity, compare against other objects, and possibly fetch a full representation; a `type`; and a `name` or `summary` (or `nameMap` or `summaryMap`) to show in a text medium. This option should let consumers show the object in a UI in a minimal way ("a note from Evan Prodromou"), as well as decide what to do with the object (such as storing it in their Persons table in a database).

Functional Representation

Next, consider which of your object's properties the consumer is most likely to use. I call this the *functional* representation, since it catches the key functionality needed by a majority of consumers. For example, here is a representation of a `likes` property for a `Note`:

```
"likes": {
  "id": "https://social.example/users/evanp/notes/1/likes",
  "type": "OrderedCollection",
  "totalItems": 25,
  "first": "https://social.example/users/evanp/notes/1/likes/page/1"
}
```

Here, I'm guessing that the consumer will want to show the total number of likes on the `Note` and maybe get each of the `Like` activities to show who liked the `Note`. Adding the first property lets the consumer know that the collection has pages (so there won't be an `orderedItems` element in its full representation) and indicates how to get the first page.

It's not easy to guess what the functional representation of an object should be. One trick is to look at various uses that consumers have for your objects. For this `likes` collection, you might look at the likes count on images in the consumer's UI and decide that including that count might save them a web request.

Link Representation

A *link* representation shows all the known properties for the given object. However, for properties with object values, you include only the `id`, not the full object. This lets consumers get the most up-to-date version of those object values directly.

This is a great representation for storing AS2 data in a database (as in the next code example), or when your application isn't sure what data it can share about remote objects. This approach trades off efficiency for brevity.

Here's a `Create` activity in its link representation:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/create/1",
  "type": "Create",
```

```
  "actor": "https://social.example/users/evanp",
  "object": "https://social.example/users/evanp/notes/1",
  "summaryMap": {
    "en": "Evan Prodromou created a note."
  },
  "published": "2023-12-05T22:30:00Z"
}
```

Full Representation

Finally, a *full* representation shows all properties of the object that your software knows about (and is willing to share with a consumer). Any properties that have object values should have a functional representation. Here's a full representation of a "Hello, World!" activity:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evanp/create/1",
  "type": "Create",
  "actor": {
    "id": "https://social.example/users/evanp",
    "type": "Person",
    "name": "Evan Prodromou",
    "preferredUsername": "evanp",
    "icon": {
      "type": "Link",
      "href": "https://social.example/uploads/evanp.png"
    }
  },
  "object": {
    "id": "https://social.example/users/evanp/notes/1",
    "type": "Note",
    "content": "Hello, World!",
    "likes": "https://social.example/users/evanp/notes/1/likes",
    "replies": "https://social.example/users/evanp/notes/1/replies",
    "shares": "https://social.example/users/evanp/notes/1/shares"
  },
  "summaryMap": {
    "en": "Evan Prodromou created a note."
  },
  "published": "2023-12-05T22:30:00Z"
}
```

You should use the full representation when responding to a GET request for an object's ID in a web API, for example, as described in Chapter 3. Another good use of the full representation is for delivering an activity over the ActivityPub federation protocol, as described in Chapter 4.

This kind of representation can be even further fine-tuned. You could include a functional representation for the values of certain properties of the object properties, like

those `likes`, `replies`, and `shares` collections. Going further down the rabbit hole, you could add the first few members of each collection:

```
"likes": {
  "id": "https://social.example/users/evanp/notes/1/likes",
  "type": "OrderedCollection",
  "totalItems": 25,
  "first": {
    "id": "https://social.example/users/evanp/notes/1/likes/page/1",
    "type": "OrderedCollectionPage",
    "orderedItems": [
      {
        "id": "https://other.example/users/chase/likes/14",
        "actor": {
          "id": "https://other.example/users/chase",
          "type": "Person",
          "name": "Chase N"
        }
      },
      {
        "id": "https://social.example/users/janet/likes/227",
        "actor": {
          "id": "https://other.example/users/janet",
          "type": "Person",
          "name": "Janet R"
        }
      }
    ]
  }
}
```

This lets the consumer add a bit of text to the UI without requiring another web hit, like "Chase, Janet, and 23 other people liked this note."

Before we go too far down this path, though, it's probably worth noting that many, maybe even *most,* consumers will ignore the extra information included in these object properties. They'll trade off optimizing runtime for optimizing developers' time. So they might check whether the `actor` property is a string, and if so, fetch its full representation; if it's an object, they'll grab the `id` property and use it to fetch the full representation, ignoring any other properties in your carefully crafted representation.

Including just the right amount of information in your object representations can help more nuanced and careful consumers save their users some time, and it can save your servers some traffic. That's usually worth taking a few minutes to consider which top properties to include.

Names

Are you feeling confused by some of the terminology I've used in this chapter? Well, trust me, you're not alone. I've used the most common naming conventions in the ActivityPub world, but some of them are really hard to keep track of. I want to highlight two of the trickiest ones.

I use *object* for at least four concepts in this chapter alone:

- A JSON object, like `{"property1": "value1"}`
- The `Object` type, which is the base type for all AS2 types
- The group of object types that represent inert, unconscious digital objects (like images) or physical objects (like places)
- The `object` property of an `Activity`

"The `object` is an object that has type `Object`" is technically a meaningful sentence but probably a head-scratcher for most readers.

I use *actor* for two things:

- The property of an activity that indicates who did the activity
- The actor types, like `Person` or `Group`

In Chapter 3, you'll see *actor* used for yet another purpose. As I've noted, actor types are often but not always the actor of an activity. And even though AS2 has an `Object` type, there's no `Actor` type. All actor types have more specific names, such as `Person` or `Group`, and all are subtypes of `Object`.

Using Activity Streams 2.0

Social web applications use AS2 in two main ways.

First, let's look at *on-the-wire usage*. An application can have its own internal representation of socially important data, like activities, actors, and objects. When it needs to communicate with another social web application, it can translate that internal representation into AS2; when it receives AS2 data from another program, it can translate that into its internal representation before acting on it.

Second, in the *native usage* paradigm, the application uses AS2 as its native data model. After receiving AS2 data from another program, the application might parse that data into an intermediate format but will work with the AS2 types and properties directly. The application stores the native model, and when it needs to communicate with other programs, it makes light modifications before sending over its internal model.

Each approach has pros and cons. Using AS2 natively makes supporting a wide array of data types much easier. But it also requires very flexible internal data types and prevents easy use of fixed field storage like a Structured Query Language (SQL) database. This approach lends itself more to NoSQL document-style storage.

Using AS2 only as an on-the-wire representation requires additional code for marshaling back and forth between the internal model and AS2. Also, the received data loses fidelity when converted into the internal format. Supporting many data types or properties is difficult in this paradigm, so most implementations just drop any types or properties they don't recognize. But internally processing code is typically simpler, since the internal representation is more fixed and rigid than the flexible AS2 style.

For existing applications or their plug-ins, on-the-wire representation is the only way to go. You need to map the AS2 versions of users, text, images, groups, and so forth in and out of your existing internal representations.

Also, if you have an application with a limited scope—say, a single-use bot or an engine for generating game achievements—AS2 can be useful as an external representation only. But for new applications intended to handle a lot of social-data types, using AS2 as your internal representation can be a good strategy. Handling all the flexibility that AS2 allows requires some discipline, but if you can manage it, you get a lot of benefit in return.

Storing AS2 Documents

Often, in ActivityPub server or client applications, it's necessary to store AS2 data in a database for long-term use. This could be for AS2 data that your application creates or for caching AS2 objects received from other servers. Different approaches have different strengths and weaknesses. This section explores the options and then takes a closer look at storing collections.

Considering Storage Options

Let's review the options for storing AS2 documents:

Existing database structure
: If you're mapping AS2 into your existing data structure and just using it as an on-the-wire format, there's not much I can tell you! Please try to use unique and persistent `id` values. If your application is caching remote data, however, you might still want to consider some of the other storage formats in this list.

Filesystem
: If you have a native AS2 application, you can store AS2 data as JSON files in your filesystem. This is also a good storage strategy for an AS2 object cache, especially for client applications. One easy format is to have a single *data* directory, with

each object's data stored in a file named with the object's `id` property. This should make it easy to quickly look up the object you need by `id`.

However, URLs as IDs aren't really great filenames, since they have a lot of conflicting characters on Unix-like systems. You might want to consider using a one-way hash to turn an ID like *https://social.example/users/evanp/notes/1* into a hash string like *15d4c260*, using a hashing algorithm like CRC-32 or SHA-256. These strings will usually have a fixed length and only filesystem-friendly characters, which results in better filenames.

One problem with storing a lot of files in a single directory is that access time can become slow when the directory contains a lot (thousands or tens of thousands) of files. You can get around this by using a tree of directories, with substrings of the filename as subdirectory names, so that the file *15d4c260.json* might actually be in the directory *data/1/15/15d/15d4c260.json*.

Key-value database
: A key-value database, like Redis, can be a great option for storing AS2 documents—especially for caching remote data. Using the `id` property as the key, and the object's link representation as the value, you can have fast lookups and updates. If your system has restrictions on key characters so that URLs won't work as keys, you can use a one-way hash, as with the filesystem approach detailed previously. This approach is great for looking up a single object, but not so good at searching for all objects that meet a particular criterion, like having the same `attributedTo` property.

SQL database
: SQL databases like SQLite, MySQL, or PostgreSQL can be a great way to store AS2 objects. You can have a simple table with an `id` column as the primary key, and a `data` column containing the JSON-LD for the object. This is practically identical to a key-value store, but you can also add metadata columns like a timestamp, or columns for particular object properties so that you can more easily do lookups by property. This is the usual approach I use for building ActivityPub applications.

Document database
: JSON-based document systems like MongoDB have advantages for storing AS2. As with key-value databases, you can map the object's `id` to its JSON-LD value. However, document databases often have mechanisms for reading, searching, or modifying parts of the JSON-LD value directly.

Search database
: A search-oriented system like Elasticsearch can be useful as secondary storage, focused on quick full-text search performance. Search engines should index human-readable text properties like `name`, `summary`, and `content`. Where

possible, including other properties as document metadata can also be useful for some types of search—say, for example, the `name` of the `attributedTo` actor. This approach can be a helpful addition to other approaches that do well with quick direct-access lookups and updates.

Triple store
: A *triple store*, a database optimized for storing Linked Data, can be a great tool for AS2. (Linked Data is detailed in "Is It JSON or Linked Data?" on page 65.) However, triple stores are often optimized for making inferences about objects and their properties, rather than fast lookups, updates, and searching. If you're used to triple stores, this is probably a great approach to try; otherwise, the learning curve might be too steep for this option to be helpful.

As I've mentioned, the best representation for storing AS2 data in a database is the link representation: all known properties are included, but properties with object values use the `id` of the object instead. This gives compact representations and doesn't cause staleness problems. On the downside, reading a single activity from the database may require fetching a half-dozen objects by their ID, so it works best with storage that allows quick direct access.

Storing Collections

One important aspect of data storage is collections. As I've mentioned, a collection is a series of elements, possibly divided into pages. These present distinct challenges for ActivityPub storage management.

One strategy is to use whatever mechanism is built into the platform for storing a series of objects. For example, in a relational database, it may make sense to represent collection membership with a table with columns like *collection_id*, *item_id*, and *order_num*. Getting the collection contents just requires getting all the membership rows with a matching *collection_id* and ordering them by *order_num*.

Collection paging, in this strategy, can work by grouping these members into subgroups of a fixed size; say, the first 20 items are on page 1, the second 20 items are on page 2, and so on. Getting the items on a page just means bookending by the right *order_num* values.

The problem comes when we realize that most important collections in ActivityPub are ordered in reverse-chronological order, based on the time they are added to the collection. The most recently added items are put at the beginning of the collection, pushing everything else further down the ordering. In other words, adding one item changes every single page of the collection, which can really mess up caching, fetching, and other access.

This option can work, but I really recommend using another strategy if possible: instead of having pages just be slices of a huge stored array, let them be first-class objects with their own contents. Adding a new element to a collection then means adding that element to the first page, and all other pages stay the same. If that page hits the arbitrary size limit (20 elements, say), create a new page, make it the first page of the collection, and stick new elements in there. Implementation-wise, this is a lot simpler. The downside of this strategy is that the first page—the most likely to get fetched—is always less than or equal to the max page size.

Is It JSON or Linked Data?

Many of the rules of AS2, and much of its structure, come from the JSON typing mechanism we use, JSON-LD. Linked Data (LD) is an architecture that gets some people really, really excited. It's a way of describing objects (or *resources*) on the web by using properties and values, just as I've done in this chapter with JSON. One neat part of LD is that the full description of an object doesn't all live in one place; it can be smeared out across many web servers, with many authors and owners.

One way that JSON-LD influences AS2 is its *compaction* rules. This is a set of steps that JSON-LD processors are supposed to follow to make a JSON-LD document uniformly terse and readable. If you use a JSON-LD-based parser, you'll likely have this handled for you.

Some parts of the compaction process are pretty intuitive: for example, it generally uses the unprefixed versions of AS2 names, or the short prefixes if they are available. Other times, compaction can cause some difficulty—for example, when array-type properties have a single element, compaction will often replace the whole array with the single value. I highlight this issue for relevant properties in Appendix B.

Most developers on the social web today don't use JSON-LD-based parsers, and their documents are not usually compacted perfectly. That's OK; they're usually quite easy to parse anyway.

JSON-LD provides a pragmatic compromise between the potentially open-ended, academic world of LD and the needs of people who want to leave work at 5 p.m. and get home in time for supper. So, for the most part, you can just treat AS2 as regular old JSON. Thinking in terms of LD is mostly necessary when you're including external vocabularies.

Conclusion

By now it should be clear how expressive and powerful Activity Streams 2.0 is. Grounded in JSON, with a simple abstract model, AS2 can represent almost any social-network content, user, or activity. And for anything that's missing, it has a robust extension mechanism to handle new or niche social concepts, as I'll explore in Chapter 5.

I hope you're excited to put these AS2 types to good use. In Chapter 3, you'll learn how to move AS2 objects between client devices and servers to give a full read-write social-network experience. And Chapter 4 will take them even further, passing activities between servers to create an integrated social web.

CHAPTER 3
The ActivityPub API

Now that you know all the ways to structure social-network data, it's time to start using it. In this chapter, I'll show you how to use the ActivityPub API, the social-network API defined in ActivityPub.

It's a strange tradition that when authors introduce a new API, they feel the need to expand the acronym to its full definition, *application programming interface*, as if that would explain what an API is and does. I find this unhelpful but also cannot resist repeating it here.

An API is an *interface* that lets programs outside the main server create, read, and change objects inside the service. The programs, or *apps*, or *applications* (see?) can be the official mobile apps for the service, or they may be created by developers working for other companies, volunteers, hobbyists, or what have you. An API is a model of how the internals of the server work, but it's also a set of permissions, defining what external programs can and can't do.

The ActivityPub API is a *web API*. That means you can interact with ActivityPub servers over the internet, using HTTP as the transport protocol. If your device has access to the web (whether via a laptop, a phone, or a toaster), it can access ActivityPub servers by using the ActivityPub API.

ActivityPub is a *RESTful API*. *REST* is a term that comes from Roy Fielding's 2000 PhD thesis about the architecture of the web. In case you don't have it on your bedside table, REST stands for Representational State Transfer (though nobody ever calls it a "representational state transferful application programming interface").

Here's the crash-course definition: when you use a RESTful API, you use HTTP to send requests and post data to particular *endpoints* provided by the API server. HTTP defines *verbs* to take actions at those endpoints: GET to get the stuff that's there, POST to create new stuff, PUT or PATCH to update things, and DELETE to, well,

delete things. A RESTful API is roughly defined by its data structures and the endpoints it uses.

If this is your first time at the RESTful API rodeo, I highly recommend the lovely and very readable text *RESTful Web APIs* by Leonard Richardson et al. (O'Reilly, 2013). He covers all the ins and outs of using RESTful APIs.

Using a Standard API

The ActivityPub API is a *standard API* (or, maybe better, an *API standard*). Experienced social software developers will note that this is kind of unusual in the world of APIs. Different software and web services often provide specific APIs for their own system, with extensive documentation that covers their specific service. They may use standards like JSON or HTTP or OAuth, and standard patterns like REST, but their data structures and endpoints are often specific to the service.

Why use a standard API for the social web? Because having different APIs for different social-network services incurs a lot of costs to app developers. First, developing and maintaining software for each of those networks takes time and money, which leads to a degree of lock-in for client developers; it's hard to move between services. And that power imbalance introduces problems for client developers. Because they can't leave, service providers may change the terms and conditions of API use at their whim, adding restrictions or even removing access. And if your chosen platform goes under, your software becomes useless overnight.

Surprisingly, having different APIs also has a big cost for service providers. You have to design, architect, deploy, and document your full API. You have to work hard to attract client developers to use the API, and if they're locked into another social-network service already, this can be really hard to do. Remember Metcalfe's law, described in Chapter 1: because client developers gravitate to big services, it's really hard to launch a new social-network service and get client developers to use it.

The standard API approach, on the other hand, helps both client developers and service providers by decoupling the API client from the API server. Client developers can develop for multiple platforms with the same codebase. They're not dependent on any single platform; they can switch over to competitive platforms without even a recompile. This approach greatly lowers the risks involved in making new client software.

Additionally, with a standard API, server developers have a broad ecosystem of client software available on day one. They don't have to do as much work in writing documentation or in evangelizing to and supporting developers. (They can even just give their client developers this book!) With luck, reducing some of the friction of launching new services can encourage innovation in social-network services.

The World of API Clients

So what exactly would you use this API *for*? The obvious answer is for general-purpose social-network clients that let users browse, post, like, and comment (Figure 3-1). You can make this kind of client as a web-based service or a desktop or mobile native app.

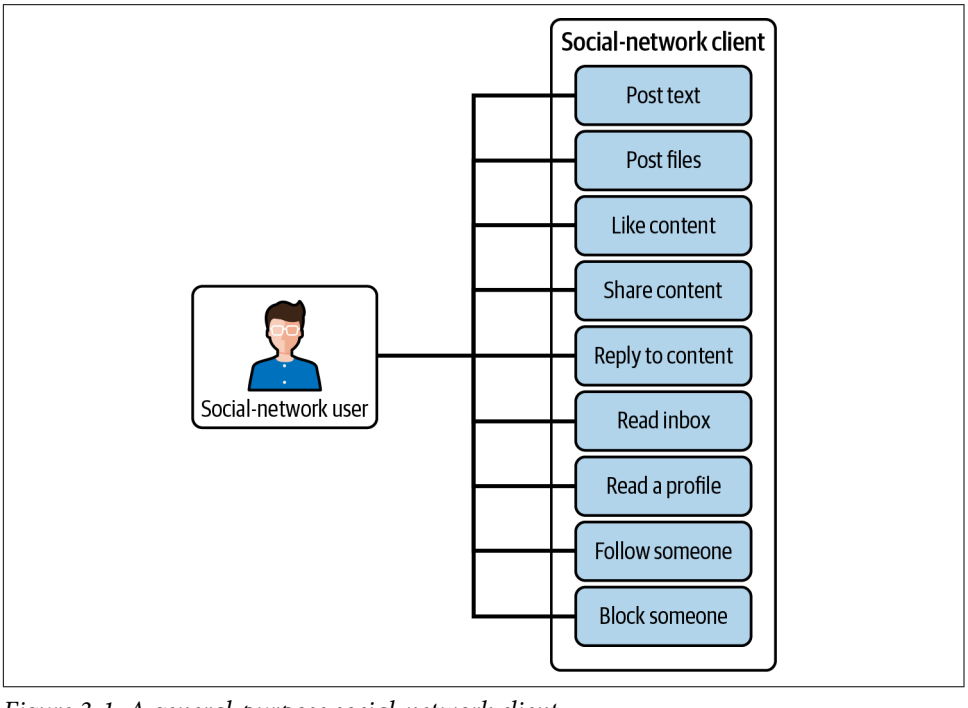

Figure 3-1. A general-purpose social-network client

But you could also use the API for social-network analysis. You could find people in your extended network whom you might want to follow, based on their self-descriptions or the number of your friends who follow them. Or you could track how often you cuss in an average month or get a map of the locations of all the people who follow you.

You could hook up an API to your electronic pet door and have it send you a message anytime your dog leaves the house to go into the backyard. Or you could post photos of the sky over your office every time the barometer drops down to the rain range.

You could make a tool that reads RSS feeds and posts the contents onto the social web. Or you could make a tool that posts a lightning citizens' poll on every decision on the agenda at your city council meeting.

People make beautiful, silly, intensely meaningful, and absolutely essential tools with API clients. That's one of the reasons we all love social networks; this worldwide community of software developers can bend the structure of a social network to a thousand purposes, a million needs.

An Extended Example

I won't be showing you any of those amazing client apps in this chapter, though. We'll get to that fun stuff in Chapters 5 and 6! For now, we're going to keep it simple, focusing on the bare essentials of using the ActivityPub API. I'll show you example code from a program called `ap` (for *ActivityPub*), which uses subcommands like `create`, `follow`, and `undo` to execute social commands. While it won't exactly win any awards for creativity, it should give you enough guidance that you can start making your own dreams happen with the ActivityPub API.

`ap` uses Python, a simple programming language that complex people love to use. It was designed as a learning language by its creator, Guido van Rossum, and consequently it is pretty easy for almost anyone to pick up. If you're unfamiliar with Python, *Head First: Learn to Code* by Eric Freeman (O'Reilly, 2018) is a great way to start. As of this writing, `ap` uses Python's latest version, 3; it won't work with version 2 and lower.

The `ap` example program is a command-line tool, and you can use its subcommands to perform social tasks with an ActivityPub API server. For example, this will post a simple, open-hearted greeting to everyone:

```
ap create note --public "Hello, World!"
```

And this will "follow" another user (in this case, me!) to start receiving updates they post:

```
ap follow evan@cosocial.ca
```

To parse these complex command lines, `ap` uses the built-in argparse library that comes with Python 3. This lets you map commands from the command line to functions in your codebase. In the following sections, I'll include the functions from `ap` that use the information shown there, at least in part. That should show you roughly how to use the API as described.

HTTP is a complex, powerful, and beautiful protocol that mortal programmers should never have to program to directly. Instead, for `ap`, I use the requests library to make all the HTTP requests needed to illustrate the ActivityPub API at work. This library offers methods that pretty directly reflect the related HTTP verbs, which return a response object that you can use to show the response. So, for example, to get the full text of the ActivityPub specification and print out its character length, you can use this Python code:

```
import requests
r = requests.get('https://www.w3.org/TR/activitypub/')
print(len(r.text))
```

Really, that's about it for understanding `ap`. The program does a little bit of config-file manipulation to, say, save the OAuth 2.0 access and refresh tokens (more about those later), but mostly what it does is command-line parsing and calling RESTful API endpoints.

One last thing about `ap`: like all software, it will change over time, and the version you find on GitHub might not exactly match the examples in this book. I promise (cross my heart) that I won't allow changes to `ap` that are unnecessary or make the code harder to read and understand.

A Follow-Your-Nose API

As a software developer, you may be used to client APIs that are structured by their URL templates. For example, if you want to get a user's photos, you might fetch */user/evanp/photos*. To get the members of a group, you might fetch */group/347/members*.

Documentation for this kind of API usually consists of a collection of URL templates with parameter specifications for each, a description of the data each endpoint will return, and the errors that will be returned if the wrong parameters are provided or something else goes wrong. Well, you're not going to get that kind of API documentation in this chapter, because that's not how the ActivityPub API works. Take a deep breath, let go of that expectation, and when you're ready, read on.

The ActivityPub API is a *follow-your-nose* API. When you start out with one resource, like an event invitation, you can use properties of that resource's representation to find related resources, like the event host, the event location, and the list of invitees. If you need to, you can keep following properties in the related resource's properties, like the event host's avatar, or the fourth invitee's first follower's eighth video's thumbnail.

This is an important distinction for an API. Sure, a lot of ActivityPub API servers will put a user's inbox at */user/evanp/inbox* and another user's fourth article at */user/katherine/articles/4*. But as a client developer, if you want to support many kinds of servers, you have to overcome the temptation to program to the URL templates that one server uses and instead follow your nose along the line from your source object, through its properties, on to other objects related to it.

Fielding had a name for this pattern in RESTful APIs: *hypertext as the engine of application state*. Here, *hypertext* doesn't mean HTML per se, but data formats like AS2 that can include links between objects. This principle is so important and long-winded that web architects have given it an acronym, *HATEOAS*. (We often

pronounce it "HATE-ee-OH-us," which sounds like a breakfast cereal nobody wants to eat. This is one reason web architects aren't allowed to name breakfast cereals.)

This kind of API has pluses and minuses. The main downside is that it can take several HTTP requests to get the exact AS2 object you want—starting at point A and fetching related resource B, then its resource C, and so on until you get to point D or F or Z. Usually, you can overcome this problem with standard web techniques like client-side caching and connection keep-alive; I'll include some in "Optimizing the ActivityPub API" on page 113.

The plus side is that client code that uses HATEOAS and follows its nose is extremely resilient to change. If you've used a single platform's API for any period of time, you have almost definitely experienced a major, breaking version change that moved the endpoints you baked into your client code, requiring you to upgrade your software. With a follow-your-nose strategy, it doesn't matter if an endpoint moves from one side of the API to the other, or even to another server, as long as the output format stays the same. So when */user/katherine/articles/4* moves to */articles/8001* or */object/40F1519A-BD26-4471-BD10-94036157B06E*, your client code can stay the same; it can still find Katherine's article through her `outbox` property.

The other advantage, of course, is that not needing to use the ActivityPub API server's URL routes to fetch the data you need means that you can develop your client for one server platform and it will work for *any other server* that supports the same standard. Following your nose requires discipline—the siren song of those easily recognized URL patterns is pretty strong—but the payoff is an app that is truly platform-independent.

Of course, it's not up to you and me to decide; the ActivityPub API standard works this way, and if we want to write ActivityPub code, we need to follow our noses. But I can assure you, it's pretty fun and pretty easy (as you get used to it), and the advantages really open up as you go along. So stop your grumbling, put down your spoon, put away your bowl of hatey-ohs, and let's get to work using the standard social-network API.

Following Rules for ActivityPub Data

As you learned in Chapter 2, each AS2 object can have a few important properties, such as `id`, `type`, and `name`. These have fairly open structures and styles for bare AS2, but if you're going to use them in an ActivityPub context, you'll need to follow some additional rules.

The most important is that the `id`'s URL has to resolve to an AS2 representation of the object. Say you see an AS2 `Follow` activity like this:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/user/edgar/follow/3",
  "type": "Follow",
  "object": {
    "id": "https://social.example/user/marcus",
    "type": "Person",
    "name": "Marcus Peabody"
  }
}
```

You can fetch the resource at the URL used for the `id` property of the `object` property to get the full information about that person:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/user/marcus",
  "type": "Person",
  "name": "Marcus Peabody",
  "image": "https://social.example/images/marcus.png"
}
```

The second restriction is that the protocol for the URL has to be HTTPS. You don't have to worry about fetching ActivityPub objects over obscure or proprietary protocols defined by the object creator; everything should be available over good old widely implemented HTTPS.

The third and final restriction is that every Activity Streams object must have an `id` and a `type` property in addition to the `name` or `summary` property AS2 requires. Overlooking the usefulness of this restriction can be easy. It means that, as a client developer, you're never more than an HTTPS request away from getting the full set of properties from a (possibly abbreviated) representation of the object.

Using these IDs as HTTPS URLs carries a few caveats. An important one is that ActivityPub API servers are explicitly allowed to use content negotiation to show different content for the same URL. *Content negotiation* is a feature of HTTP that lets the HTTP client define a list of media types it will accept for the request, including rough prioritization on a scale from 0.1 to 1.0. The server can then decide which of the URL's representations of the resource best meets the client's needs and return that one.

For example, in the ActivityPub world, it's common to use the same URL for the AS2 representation of a person and for that person's HTML web page. Let's say a Python client requests such a URL at *https://social.example/user/greta* by using the requests library and outputs the `Content-Type` header value. The requests library keys to this header value by its lowercase name:

```
import requests
r = requests.get('https://social.example/user/greta')
print(r.headers['content-type'])
```

This may very well print out `text/html`, the internet media type for HTML pages. Because you didn't specify which data format you want, the server returns its default type for this resource: HTML. Instead, you need to specify an `Accept` header indicating that you want AS2 data. AS2 has three main media types, as discussed in Chapter 2, so you can include those in your header. Here's the resulting Python:

```
import requests
headers = {
  "accept": "application/ld+json,application/activity+json,application/json" }
r = requests.get('https://social.example/user/greta', headers=headers)
r.raise_for_status()
print(r.headers['content-type'])
```

Now, since you told the server that you want it to return the resource in only one of these formats, it will return the best match for the resource. Usually, for ActivityPub servers, the "right" version is `application/ld+json`.

Note the line with `r.raise_for_status`. That's what we use to tell the requests library to throw an exception when an HTTP request fails. In this case, a common failure status might be `406 Not Acceptable`, which is what servers use to tell us that they can't actually return the resource we asked for in any of the formats we specified. In the ActivityPub world, that's unusual, but it sometimes happens when you ask for an image URL or an HTML profile page that isn't an actual ActivityPub ID.

Another caveat is that, as a publisher, you can't use the same `id` value for two different objects. That may sound obvious, but this rule can be quite tricky as time goes by. For example, if one actor follows another, it may seem reasonable to give the `Follow` activity an `id` that includes the follower and the followee in the URL, like this:

```
https://social.example/users/greta/follow/other.example/users/mau
```

But what if Greta decides to take a break from following Maureen for a few months, and then comes back and follows her again? By this URL pattern, the second `Follow` activity would have the same `id` as the first—explicitly not allowed. You can use tricks to get around this, like including a datetime stamp in the URL, or a random value.

Another issue is that you simply can't change the `id` of an object—not an actor, not a note, not a `Like` activity, nothing. Therefore, any properties of the object that could change over time shouldn't be included in the URL. For example, many of the URLs I used for `id` properties in this book include what looks like a username or nickname in the URL (like the *greta* in `https://social.example/users/greta`). But including this username makes it hard for Greta to change her username if she wants to. Even if the system allows it, it would be a problem if a new user took that old nickname. (I still like this URL format, though, since it can convey a lot of information to someone reading the URL. And my imaginary actors lack the free will to change their nicknames, so the risk is much lower.)

On a server-wide level, the domain name part of the URL can't change either. This is often a surprise for server operators who set up their servers at *test.social.example* or something similar and then find out they can't easily change to *social.example* once their server is more stable.

Having IDs that are dereferenceable URLs also means some kind of object storage solution that allows looking up even years-old activities and objects. That can quickly get unwieldy without careful database design.

Lastly, and probably most important: a robust network means having fallback behavior when partners on the network stumble and fall. Some services might not use usable URL formats, or they may return `404 Not Found` when the URL is dereferenced. Other services might go out of business, and their servers may stop responding to ActivityPub requests. Try to make your software robust to problems with the URL value of an `id` property.

Reading Data: The Actor

Almost everything in the ActivityPub API comes down to the user. That seems appropriate, right? Social networking is about people, and we want to put our users right in the center of our process.

With ActivityPub, we almost always start with retrieving the actor object that represents the current user. An *actor* is simply an AS2 object, served from an HTTPS URL, with a few specific properties that we can use for further processing. Users are actors in the ActivityPub world, but an actor doesn't *have* to be a person (of type `Person`). It can also be one of the other actor types from Chapter 2, like `Group`, `Service`, or `Organization`. But it can also be an `Image`, an `Event`, or a `Document`. What makes actors actors is that they generate objects with an `Activity` type. There's a saying that "writers write"; in ActivityPub, actors act.

Here is sample code for fetching an actor, given its ID URL:

```
import requests
import json
headers = {
  "accept": "application/ld+json,application/activity+json,application/json" }
r = requests.get('https://social.example/user/greta', headers=headers)
r.raise_for_status()
j = r.json()
s = json.dumps(j, indent=2)
print(s)
```

This code looks a lot like the preceding example, but we've added a couple of more steps. We get the JSON value by calling `r.json`, format it with the Python `json` module, and use the `print` function to print it to the console.

What we get back are some of the normal properties you'd expect to describe an AS2 object—the `type`, the `name` or `nameMap`, the `icon` to use when displaying the actor on a screen, and maybe a `summary` bio of the actor. But we'll also get properties that are specific for actors in ActivityPub. These are trails that we can follow to learn more about the actor and work with the actor object. Here are some of the major ones:

`outbox`
: The `outbox` property is an `OrderedCollection` that returns all (well, almost all… see "Understanding the Authorization Model" on page 111) of the activities that the actor has ever done. The activities are sorted in reverse-chronological order, with the newest ones first.

`inbox`
: The `OrderedCollection` that shows all the activities that have arrived for this actor. It is their *home feed*, where the activities of every other actor they follow appears.

`followers`
: The `OrderedCollection` of everyone who follows this actor.

`following`
: The `OrderedCollection` of everyone that this actor follows.

`liked`
: The `OrderedCollection` of every AS2 object this actor has liked.

`endpoints`
: A collection of API endpoints for initiating other interactions.

Most of the interesting work in creating an ActivityPub application is done by reading these collections and adding to them, either directly or indirectly.

When you're using the properties of an actor object, remember that object properties can be represented in several ways. The `following` property might be just a single string, the ID of the `OrderedCollection`:

```
"following": "https://social.example/users/greta/following"
```

It can also be a brief representation of the `OrderedCollection`, as mentioned in Chapter 2:

```
"following": {
  "id": "https://social.example/users/greta/following",
  "nameMap": { "en": "People Greta is following" },
  "type": "OrderedCollection"
}
```

The property might even include some or all of the items in the collection, to save you an HTTP request:

```
"following": {
  "id": "https://social.example/users/greta/following",
  "nameMap": { "en": "People Greta is following" },
  "type": "OrderedCollection",
  "Items": [
    {
      "type": "Person",
      "id": "https://photo.example/users/adele",
      "name": "Adele Peterson"
    },
    {
      "type": "Person",
      "id": "https://social.example/users/tyler",
      "name": "Tyler Phipps"
    }
  ]
}
```

How these properties are represented is up to the server developer to decide. As you move between server platforms, you'll see different representations. It's important to be ready for at least brief objects and ID strings, but watching for more extended representations can save time (which is why the server developer showed it to you in the first place).

The `endpoints` array, on the other hand, contains only strings for the URLs to use for protocols loosely related to ActivityPub. I'll go into a few of them next for OAuth 2.0, proxy requests, and file uploads, but others are in the ActivityPub spec itself. Extensions can also define new endpoints to use.

WebFinger for Discovery

Fetching actor information requires the actor's ActivityPub ID, which is an *https:* URL that returns an AS2 representation of the actor. However, many fediverse services use a different representation for actors called WebFinger. *WebFinger* is a protocol to provide metadata for people—like their names, avatars, descriptions, and the endpoints they use to interact with various APIs and internet protocols, just as in the preceding examples. WebFinger is an earlier standard for getting this kind of information.

WebFinger was originally developed by technologist Blaine Cook, former chief engineer at Twitter and one of the early architects of OAuth (discussed in "OAuth 2.0 for Access Control" on page 80). Today, WebFinger is used for several distributed identity systems, like OpenID Connect, and is popular with distributed social networks as a kind of meta-identity framework. The name comes from a *very* old internet protocol called *finger*, defined in RFC 742 in 1977. WebFinger itself was defined by a loose

group of technologists in the mid-2000s and was standardized at the IETF as RFC 7003 in 2013.

There are two main differences between WebFinger and the ActivityPub actor ID. First, WebFinger identities look like email addresses (*name@domain.example*), whereas ActivityPub actor IDs usually look like URLs (*https://domain.example/user/name* or something similar). Importantly, there's no standard indicating how the actor IDs are formatted; they can have any kind of path the implementer or domain owner wants. But WebFinger is always in *name@domain.example* format—so it's easier to say out loud, to type, and to remember.

The other difference is that WebFinger uses a different dialect of JSON for encoding data. Like a small cluster of other IETF discovery standards, its data format is the JSON Resource Descriptor (JRD) schema. JRD has only a few important types; most of the semantic meaning in JRD documents comes from the choices of links and other data in the JRD clauses.

In the ActivityPub world, we use WebFinger to provide an email-like identity that can be used to discover a user's ActivityPub actor ID. As a secondary use, many ActivityPub implementations mirror some or all of the data from the actor resource to the WebFinger endpoint.

The JRD document that represents an actor with the ID *user@domain.example* can be retrieved at the URL *https://domain.example/.well-known/webfinger?resource=acct:user@domain.example*. In Python, we would do this conversion as follows:

```
def webfinger_url(wf):
    [username, domain] = wf.split('@')
    base = f'https://{domain}/.well-known/webfinger'
    return f'{base}?resource=acct:{wf}'
```

We'd then use `requests` to fetch the contents of that URL and find the link to the ActivityPub actor ID.

In WebFinger, related links are in the `links` property of the JRD document. The one we want has the `rel` (relationship) property equal to "`self`" (meaning it's another way of identifying the person with this WebFinger) and the `type` property `application/activity+json` (one of the ways of identifying AS2 data). Here's some Python code to do all this:

```
**
import requests

AS2_TYPES = [
  'application/activity+json',
  'application/ld+json; profile="https://www.w3.org/ns/activitystreams"'
]

def webfinger_to_activitypub(wf):
```

```
    url = webfinger_url(wf)
    r = requests.get(url)
    r.raise_for_status()
    j = r.json()
    links = list(filter(lambda l: l['rel'] == 'self', j['links']))
    links = list(filter(lambda l: l['type'] in AS2_TYPES, links))
    if len(links) == 0:
       return None
    else:
       return links[0]['href']

print(webfinger_to_activitypub('evan@cosocial.ca'))**
```

This code creates the URL as previously and uses requests to fetch the JRD document for the user. It then filters the `links` property of the JRD to find the right link and returns the first matching value. WebFinger explicitly defines the order of links as communicating their priority. So taking the first one is probably the best one. In practice, for ActivityPub servers, it's rare to see multiple matching links.

This code is mostly to show you how WebFinger works. WebFinger is a widely implemented protocol, and as with most protocols, using an existing library for the implementation is better than rolling your own. For Python, the `webfinger` module can take care of some of these discovery steps for you:

```
from webfinger import finger

AS2_TYPES = [
  'application/activity+json',
  'application/ld+json; profile="https://www.w3.org/ns/activitystreams"'
]

def webfinger_to_activitypub(wf):
   jrd = finger(wf)
   links = list(filter(lambda l: l.get('rel') == 'self', jrd.links))
   links = list(filter(lambda l: l.get('type') in AS2_TYPES, links))
   if len(links) == 0:
       return None
   else:
       return links[0].get('href')

print(webfinger_to_activitypub('acct:evanprodromou@evanp.me'))
```

Note that the Python `webfinger` module requires that WebFinger IDs use the `acct:` prefix, which is part of the WebFinger standard; this technically makes the WebFinger ID into a URL.

Users of ActivityPub services really like WebFinger IDs, so this transformation happens a lot in ActivityPub clients. There's something really natural about identifiers that look like "(the person) at (the organization)." We're all familiar with this pattern from internet email, so it translates well to the fediverse. Many clients display user

IDs in WebFinger format and allow users to enter WebFinger IDs anytime they refer to other users.

OAuth 2.0 for Access Control

As you were reading about how ActivityPub IDs can be used to retrieve a full description and some contents of ActivityPub objects, you might have thought, "Wow. I really don't want just anyone fetching my posts to the social web." Fortunately, ActivityPub servers agree, so most include a system of *authorization* to check who can read an object and who can't. The usual method here is OAuth 2.0.

OAuth is a system of authorization that lets a user grant an external piece of software limited access to their data. It originated in the mid-2000s, with the same loose set of technologists as WebFinger, and was later standardized at the IETF. Today, OAuth 2.0, the most recent version, is the main way that servers protect APIs. It's a natural model for API access, and many client- and server-side libraries support the standard.

OAuth has a lot of moving parts, but at its base it has five essential steps:

1. The client sends the user to the API server.
2. The API server shows the user information about the client, the client's author, and the data it wants to access.
3. If the user agrees, the server sends the client an intermediate token.
4. The client sends the intermediate token back to the server via a different route and receives a real access token.
5. The client includes the access token in the `Authorization` header of any HTTP requests it makes to the server from then on.

Fortunately for us, a great Python library is available for OAuth apps, called requests_oauthlib. It works with our HTTP requests library (lucky us!) and makes the OAuth process much easier. For the rest of this chapter, I'm going to punt on the OAuth negotiation process; you can use the `ap login` command to see the steps necessary for getting OAuth authorization.

On the server side, there are also great libraries for implementing OAuth 2.0. Many server-side authentication and authorization frameworks support OAuth 2.0, like Passport or NextAuth.js in Node.js, or FastAPI for Python.

Certain servers and services specifically support API authorization, including OAuth 2.0. Keycloak is an open source server for authorization. Amazon Web Services, Google Cloud, and Microsoft Azure also have tools for wrapping an API with OAuth 2.0; check documentation for details.

Your server needs to support three OAuth 2.0 features to work well with ActivityPub clients:

Authorization server metadata
: This framework, defined in RFC 8414, provides a method for client software to find important endpoints on the server, as well as requirements for interaction.

Dynamic client registration
: OAuth 2.0 clients use a *client ID* to identify themselves to the OAuth server. In centralized services, this ID is usually obtained through a developer portal, but for a standard API like ActivityPub, the client developer cannot register on thousands or tens of thousands of portals. RFC 7591 provides a standard way to get a client ID dynamically.

ActivityPub-specific scopes
: *Scopes*, in OAuth parlance, are the kinds of functionality that an application can use on the server. FEP d8c2 defines a set of scopes that can be used for the ActivityPub API: `activitypub:read:all` is for reading any data; `activitypub:write:all` is for creating any kind of activity (see "Writing Data: Activities as Commands" on page 87). More-specific scopes restrict reading or writing to certain conditions or data items.

There is a whole lot more to it than that, but OAuth is sufficiently deep that whole books cover just this topic. I really like Ryan Boyd's *Getting Started with OAuth 2.0* (O'Reilly, 2012) and the slim *OAuth 2.0 Simplified* by Aaron Parecki (Lulu Press, 2018).

Reading Data: Collections

Now that you have an access token, what can you do with it? The first thing that every social-software user wants to do is read their home feed and see all the cool activities that their friends, family, neighbors, and colleagues have been doing. So let's start there!

The Inbox and Outbox

As I've mentioned, the `inbox` property of the actor is an `OrderedCollection` containing activities from the actor's social network. However, this could be a bare URL; a brief representation with just a name, ID, and type; or a more complete representation with the properties we need to fetch the included activities, such as `items` or `first`.

We'll save the last possibility for a future optimization and go for just the first two right now. Let's use this utility function to turn an actor's property into an ID, whether or not it's a string:

```
def to_id(prop):
   if isinstance(prop, dict):
       return prop['id']
   elif isinstance(prop, str):
       return prop
   else:
       raise Exception('Invalid property type')
```

Now read the inbox:

```
r = requests.get(id)
r.raise_for_status()
json = r.json()
inbox_id = to_id(json['inbox'])
```

Of course, then we have to figure out what to do with the `inbox_id`!

If we get the object from the server, we have a few possibilities. First, the object could contain an `orderedItems` or `items` property:

```
{
  "@content": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/user/evan/inbox",
  "type": "Collection",
  "orderedItems": [
    "https://social.example/activity/13",
    "https://photo.example/create/photo/22",
    "https://social.example/user/fran/like/25"
  ]
}
```

The object could contain a `first` property, pointing to the first page of the collection:

```
{
  "@content": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/user/evan/inbox",
  "type": "Collection",
  "first": "https://social.example/user/evan/inbox/5024"
}
```

To get all the items, we'll need to get the `first` page, read all its items, and then get its `next` page, if any, and so on until there aren't any more `next` pages. (I wasn't joking when I said ActivityPub was a follow-your-nose API.)

Python has a nice abstraction called a *generator* that lets us go through these two types of collections and hide the implementation details from calling code. Here's how it might look:

```
def items(url):
   headers = {
       'Accept': 'application/ld+json,application/activity+json,application/json'
   }
   r = requests.get(url, headers=headers)
   r.raise_for_status()
```

```
    coll = r.json()
    if coll.get('items', None) is not None:
        for item in coll['items']:
            yield item
        return
    elif coll.get('orderedItems', None) is not None:
        for item in coll['orderedItems']:
            yield item
        return
    elif coll.get('first', None) is not None:
        page_id = to_id(coll['first'])
        while page_id is not None:
            r = requests.get(page_id,headers=headers)
            r.raise_for_status()
            page = r.json()
            if page.get('items', None) is not None:
                for item in page['items']:
                    yield item
            elif page.get('orderedItems', None) is not None:
                for item in page['orderedItems']:
                    yield item
            else:
                raise Exception('No items found in page')
            if page.get('next', None) is not None:
                page_id = to_id(page['next'])
            else:
                page_id = None
```

We can call the function like this:

```
for item in items(id):
    print to_id(item)
```

That's just going to show the ID of the activity, though. Not a great reading experience! Instead, let's try to give a little more information with a helper function. Here's one way to format an activity:

```
def format_activity(activity):
   act = to_object(activity)
   actor = to_object(act['actor'])
   object = to_object(act['object'])
   return f'{to_string(actor)} ({to_id(actor)} {act['type']} {to_string(object)}'
```

We can do this for the inbox or the outbox. With our helper functions, our `outbox` command will look like this:

```
def outbox():
    id, token = saved_data()
    actor = get_actor(id)
    outbox = to_id(actor['outbox'])
    for item in items(outbox):
       print(format_activity(item))
```

Collections like `inbox` and `outbox` are common in ActivityPub; we can use this pattern for other collections as we come to them.

The Social Graph

We have two main collections for the social graph: `following` and `followers`. They model a one-way social connection, although social networks that require two-way social connections can use the same collection for both properties.

Figure 3-2 shows a small social network of a few people. Arrows show the direction of following; arrows with heads on both ends are bidirectional follows. In this network, Alice is following Bob, Carla, Frank, and Gina. Her followers include Carla, David, Eleanor, and Frank. The two collections are shown on the right.

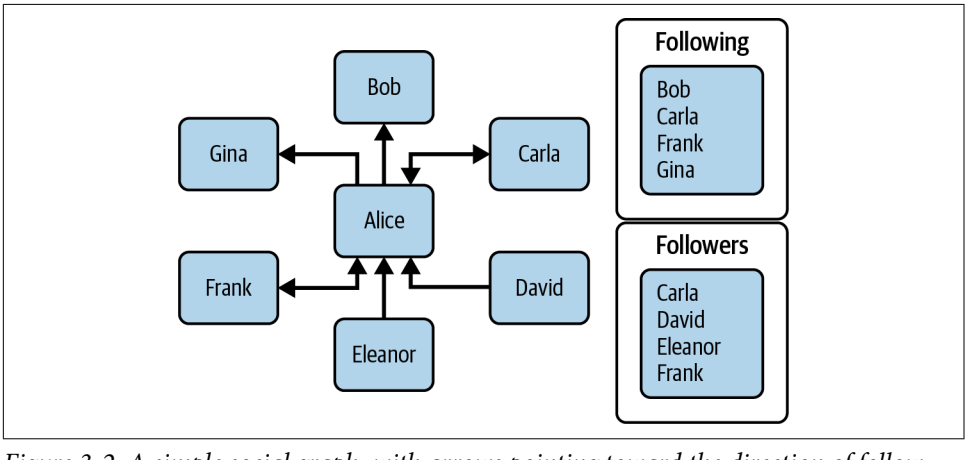

Figure 3-2. A simple social graph, with arrows pointing toward the direction of following; the `following` and `followers` collections for Alice show the edges that point out of, and into, her node in the graph

Unlike `inbox` and `outbox`, social-graph properties don't contain activities. Instead, they contain other actors to which the actor is connected. So, our `followers` command might look like this:

```
def followers():
    id, token = saved_data()
    actor = get_actor(id)
    followers = to_id(actor['followers'])
    for item in items(followers):
       print(format_actor(item))
```

The `format_actor` function might look something like this:

```
def format_actor(actor):
    id = to_id(actor)
    full_actor = get_actor(id)
    print(f'{full_actor.name} ({id})')
```

The `following` method would be about the same, except with the `following` property.

Reading Remote Data: The proxyUrl Endpoint

One part of using a federated social web is that items you might be interested in are often stored on a different server. For example, Leah's inbox might be full of interesting video, text, audio, and images, some of which is stored on another server. How can her client application retrieve it?

Let's say Leah (*https://social.example/users/leah*) gets an activity like the following one in her inbox. Take a moment to read it over. What part of this activity would Leah want more info about?

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://photos.example/users/murray/create/17",
  "actor": {
    "id": "https://photos.example/users/murray",
    "name": "Murray Hoeden",
    "type": "Person"
  },
  "to": "https://photos.example/users/murray/followers",
  "type": "Create",
  "object": {
    "type": "Image",
    "summaryMap": {
      "en": "Photo of my friend Leah at the picnic last weekend"
    },
    "id": "https://photos.example/users/murray/photo/4225"
  }
}
```

Leah wants to get more information about the image of her that Murray has posted. How should she do that?

Normally, she might try an HTTP GET (or get her client to do it for her) on the image URL. But Murray has posted the photo only to his followers (see the `"to"` line there?), not to the public. Leah will need some way to identify herself as one of Murray's followers in order to GET the image.

Can she use the OAuth authorization token that she used to log in? After all, that's what she uses to retrieve private information hosted at her own server. However, that

token is *only* for resources on her own server. Other servers are not aware of that token at all and probably can't even tell that it's Leah's.

ActivityPub has a solution for this: the `proxyUrl`, a special endpoint that Leah can use on her own server to request information on other servers. Her server requests the information from the remote server by using a server-to-server authentication system called HTTP Signatures (see Chapter 4 for how this works). Leah's server then returns it to her.

The `proxyUrl` is useful, but it's also optional; not all servers support it. So it's one of the `endpoints` in the actor object. We can get the value of the endpoint like this:

```
actor = get_actor(id)
proxyUrl = actor.get("endpoints", {}).get("proxyUrl", None)
if proxyUrl == None:
   print("No proxyUrl endpoint")
```

Once Leah has the URI, she can send a POST request to it by using her OAuth 2.0 bearer token for authorization. The `id` of the object she wants to fetch is the `id` parameter of the POST request:

```
headers = {
  "Authorization": f'Bearer {token}',
  "Accept": "application/ld+json,application/activity+json,application/json"
}
data = { "id": imageId }
r = requests.post(proxyUrl, headers=headers, data=data)
r.raise_for_status()
image = r.json()
```

Now Leah's client can look through that returned JSON payload for more information about the image. If it includes a URL, she can fetch the image the same way:

```
url = image['url']
headers = { "Authorization": f'Bearer {token}' }
data = { "id": url }
r = requests.post(proxyURI, headers=headers, data=data)
r.raise_for_status()
image = r.content # show this to the user
```

Using a POST request to fetch data definitely isn't immediately clear, but it's a good way to make sure that data on other servers is available to the user.

Leah might be tempted to have her server cache the data in the JSON payload or the binary image file. That could be troublesome, since her server would now be responsible for enforcing Murray's distribution rule (namely, that this post is available to only his followers). So, given an ActivityPub object ID, if she wants to get the JSON-LD representation of the object, when should Leah use the `proxyUrl`, and when should she just fetch with the OAuth 2.0 bearer token?

As with most things on the web, the answer is to separate the URLs by *origin*—that is, the protocol, hostname, and port. So if the user's ActivityPub ID is *https://social.example/user/evan* and the resource's ID is *https://social.example/user/katherine/note/7*, the origin for both would be *https://social.example*, and Leah should use the OAuth 2.0 bearer token and a simple GET. But if the user's ID is *https://friends.example/user/cody* and the resource is *https://photos.example/picture/64E0108D-0F2C-41E9-AC56-8CF9CCF94530*, the origins are *https://friends.example* and *https://photos.example*. Since the protocol must be *https://* and using nonstandard ports is rare outside of developer environments, the difference will probably just be the hostname.

Here's how to implement the `get` command in `ap`:

```
from ..utils import saved_data, origin, get_by_id, get_by_proxy

def get(id):
    """Get an item by ID

    Args:
        id (str): The ID of the item to get
    """

    user_id, token = saved_data()

    if (origin(id) == origin(user_id)):
        json = get_by_id(id, token)
    else:
        json = get_by_proxy(id, token)

    print(json)
```

Writing Data: Activities as Commands

Reading data from the ActivityPub server is pretty cool; we can get a lot of information from the API with just a little bit of input. But a real API needs to let you create new content and respond to it.

ActivityPub does this a little differently than you might be used to from other APIs, but the process becomes familiar over time. Remember the `outbox` property of the actor object? It's the collection of all the activities that the actor has made, in reverse-chronological order. When a user does something new, we should see a new activity in that collection. In fact, it should be the first activity, since it would be the most recent.

With the ActivityPub API, we make that happen by putting the activity there directly. It's a pattern used in RESTful APIs called *POST to create*.

POSTing to Create

When we want to add an element to a collection, we POST a representation of that element to the collection's URL. The HTTP response is the new element, along with any properties added by the server, like IDs, timestamps, or other metadata.

Posting activities can have side effects: you can modify the social graph, create text and other objects, and create reactions to content, all by posting activities to the outbox. Not all activity types will have side effects, but the ones defined in the following code will.

This can be a conceptual challenge for some developers. When a user posts an activity of type `Create` and an object of type `Note`, is that a statement of a fact that has taken place ("I have created this note") or a command for the server to execute ("Create this note for me")? I like to think of the POST to the `outbox` collection as a *performative utterance*, like making a promise, casting a spell, or knighting a loyal retainer. Just by saying the activity in the right way in the right context, you do the activity.

Only you, the user, can post to your outbox, because POSTing an activity to the outbox creates activities for that user. You need an OAuth 2.0 token before posting; that's how the server will know who you are and that you're authorized to add the activity.

Here's what the helper function `do_activity` looks like in the `ap` program:

```
user_id, session = saved_data()
headers = {
    'Content-Type': 'application/ld+json; ' +
      ' profile="https://www.w3.org/ns/activitystreams"'
}
actor = get_actor(user_id)
outbox = to_id(actor['outbox'])
r = session.post(outbox, headers=headers, data=json.dumps(activity))
r.raise_for_status()
return r.headers.get('Location')
```

The function gets an OAuth2Session and determines the correct outbox URL to send the activity to. It then sends a request with the session, which will include authentication information, to the server. The server will return the location of the new activity in the `Location` header. Many servers will also return the full body of the activity in the body of the response, but this isn't strictly required, so I've left it out of this example.

POSTing the `Activity` to the actor's `outbox` adds the activity to the outbox; that's the POST-to-create pattern at work. But a lot more happens when an activity is created!

First, the activity may have side effects—such as creating a new object, altering the social graph, updating collections, or adding to a list of reactions. This all depends on

the type of activity that was created. The ActivityPub server is responsible for recognizing the activity type, and taking the appropriate action based on the type.

The ActivityPub server is also responsible for distributing the activity to all the addressees. As you saw in Chapter 2, every actor listed in the `to`, `cc`, `bto`, or `bcc`, properties is an addressee of the activity. Because the addressees can be individual actors *or* collections, that can be a lot of work. Some collections on the fediverse have tens of thousands or hundreds of thousands of members, and every one needs to have the activity delivered.

Delivery can be *local* to the ActivityPub server for other actors with accounts on the same server. Usually, the ActivityPub server takes care of that delivery internally—by inserting records into a database, for example. But delivery can also be *remote*, to actors on other servers. And those other servers might cause even further side effects when they receive the activity. That's where things get exciting! We'll go deeper into remote delivery and how it works in Chapter 4. For now, the important thing to remember is that it's up to the client to tell the server to whom the activity is addressed, and it's up to the server to make sure that the activity gets delivered.

Handling Errors

What if an error occurs? The client might be posting a malformed activity or might be posting to the wrong `outbox`. The account might be deleted or restricted by the server administrators. Or bad state might be on the server side—a database server is down for maintenance, or the internal network in the server cluster is overloaded.

HTTP status codes are three-digit numbers; the first digit defines a related class of statuses. Here's what to do depending on which class of status code you receive:

2xx
: These are success codes. The required response is `201 Created`, but some servers may return `200 OK`. Both should include the activity in the body of the response.

3xx
: This group of status codes is for redirecting to another URL. This is an unusual response, especially if the `inbox` endpoint was just requested. It's OK to try to follow the redirects (Mozilla Developer Network, or MDN, has great information on the details of redirection), but it's also OK to just give up and treat this response as a suspicious error.

4xx
: This group is for client-side errors; your request is incorrect in some way. The `401 Unauthorized` means that your OAuth token is expired or invalid; it may be possible to recover by logging in again.

`404 Not Found` and `410 Gone` mean the endpoint doesn't work anymore, possibly because the user account is deleted or disabled. Uh oh!

`405 Method Not Allowed` means that this server doesn't allow posting activities to the `outbox`. This is kind of a bummer; either the server is read-only or it uses a custom API, separate from ActivityPub.

`429 Too Many Requests` means you've hit an API rate limit on the server. The `Retry-After` header can give a hint of when to try again.

Other *4xx* codes are a sign that your client code has a bug; it probably makes sense to log the error and possibly report a bug.

5xx

This group is for server malfunctions. They may have a `Retry-After` header, suggesting that you can retry the request. Otherwise, it probably makes sense to just log the error.

Error responses don't have a standard format in the ActivityPub API, so it's kind of up to you and your HTTP client library to decipher what comes back. RFC 7807 defines a standard JSON format for error responses, but it's not universally implemented by servers. Some servers return plain text, HTML, or custom JSON formats. Sorry about that!

Making Things

In many web APIs, the most important functionality is summed up in the acronym *CRUD*: create, read, update, and delete. You've just learned how to read an ActivityPub object by fetching its ID, so let's look further into other parts of the object lifecycle.

Create

The `Create` activity is one of the most important tools in the ActivityPub developer's toolbox. We use it to create new objects. Here, for example, is the JSON-LD for a `Create` activity to create a well-known universal greeting:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "to": "as:Public",
  "object": {
    "type": "Note",
    "contentMap": {
      "en": "Hello, World!"
    }
  }
}
```

Here's the Python code to submit this activity:

```
activity = {
  "type": "Create",
  "object": {
    "type": "Note",
    "contentMap": {
      "en": "Hello, World!"
    }
  }
}
result = do_activity(activity)
print(result)
```

The results returned to the client will usually have the full set of information for the activity, including IDs, reaction collections, and timestamps. This is particularly important for the `object` property's `id`. If you might want to modify or otherwise manipulate that object in the future, keep the `id` somewhere so you can reuse it.

Issues to watch out for

The standard has some conceptual trickiness here. When we `Create` an object, does that actually, well, create the object? Sometimes yes and sometimes no. A good rule of thumb is that if the object already has an ID, you can assume it already exists externally somewhere else. The following activity would notify an actor's followers that the actor has created a `Barn` object in a (fictional) farming-simulation game:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://namespace.example/games/farm"
  ],
  "type": "Create",
  "to": "https://social.example/users/evan/followers",
  "object": {
    "id": "https://farmgame.example/barn/223",
    "type": ["Barn", "Object"],
    "nameMap": { "en": "Barn 223" }
  }
}
```

Note that the barn already has an ID, managed in a server separate from the user's main account server.

Another detail to watch for is document types that have a `url` property, like `Image` or `Video`. This typically means that the data is hosted outside the main account server:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "to": "https://social.example/users/evanp/followers",
```

```
  "object": {
    "type": "Image",
    "url": "https://photos.example/photos/100"
  }
}
```

These are useful patterns for applications that use ActivityPub to distribute notifications to the user's social network. However, a downside exists: the client application doesn't have the up-to-date, complete information about the user's authorization groups, like their followers. So, it can be hard for these servers to enforce access control. In general, it's safest to manage the binary representation (the file) and the structured representation (the AS2 object) on the main account server, where information about the followers and other groups is stored.

Microsyntax

Many social-network clients support a textual shorthand for creating links, tags, and mentions derived from years of tradition in blog comments, chat services, and microblogging platforms. Called *social microsyntax*, it is not formally defined as of this writing but includes the following transformations of text patterns:

@username@domain.example
: A mention of the user with this WebFinger address. A typical treatment is to replace the string with a link to the user's profile page, if available. This should also generate a `Mention` tag for the mentioned user and include them in the `to` addressing property.

@username
: Similar to the full WebFinger example, but for a user on the author's local domain. If the author is on *social.example, @username* would link to the actor at *username@social.example*.

#hashtag
: This pattern is replaced with a link to a list of objects with this same hashtag. Hashtags are an easy way to create bottom-up categories covering particular topics, content types, points of view, or other content characteristics. When this pattern is found, a `Hashtag` object is included in the `tag` property of the object.

https://example.com/
: URLs in source text are often replaced with a link to the URL, with the URL as the text.

example.com
: Bare domain names in the source text are sometimes replaced with links to the home page for that domain name, like *https://example.com/*. However, matching domain names is a little tricky, since some filename extensions are also top-level

domains, and since you don't want to double-encode the domain names in URLs or WebFinger addresses.

:emoji:
: This pattern is sometimes replaced with a Unicode emoji character with a matching name. It can be hard to do this replacement well, since the proper replacement character is culture- and language-dependent. Good libraries can help manage this complexity; Python has the emoji package, and Node.js has node-emoji.

These transformations are usually applied only to the `content` property of a `Note`, but some clients also transform the `summary` property of document-like objects like `Image`.

The `source` property is included in an object to make it easier to edit later. This property is an object; it should include a `mediaType` property and a `content` property. For microsyntax, the media type is `text/plain`.

Here's a snippet from `ap` that creates a note based on input text from the command line:

```
obj = {
    "type": "Note",
    "source": {
        "mediaType": "text/plain",
        "content": self.source
    },
    "tag": self.getTags(self.source),
    "content": self.sourceToHTML(self.source)
}
```

This code adds the `source` property to the note. It also looks for microsyntax that should generate a tag and uses that for the `tag` property. Finally, it sets the context property to the HTML derived from the source. This code is a little inefficient (it is matching mentions and hashtags twice), but I'm making the trade-off for clarity.

Here's the `sourceToHTML` method:

```
def sourceToHTML(self, text):
    html = html.escape(text)
    re.sub(URL_PATTERN, lambda m: self.link(m.group()), html)
    re.sub(r'\b@\w+@[\w.]+\b', lambda m: self.mention(m.group(), html)
    re.sub(r'\b#\w+\b', lambda m: self.hashtag(m.group()), html)
    html = emoji.emojize(html)
    return html
```

Here, the first step is using the built-in `html` package in Python to escape important characters in HTML like `<`, `>`, and `&`. This is important even if you don't want to support social microsyntax.

Next, the code detects matches to the pattern `URL_PATTERN` (which is long and complicated!), and replaces those with a link by using a method of the command object. It does the same with mention patterns and hashtag patterns. Finally, it uses the `emoji` package to replace the emoji pattern with emoji characters. Note the order of the replacements: it's better to replace URLs first, so the regular expression won't overmatch on mention or hashtag links.

Microsyntax is an important part of the social-network user experience, especially when you provide plain-text input formats. The regular expressions can be a bit maddening, but the satisfaction from users when they can link and organize content is usually worth it.

Update

What about changing objects? That's where the `Update` activity type comes in. It takes an *existing* object as its `object` property. *Every* property of the JSON object in the activity's `object` property overwrites the existing properties of the object on the server.

Let's say we create a `Note` object, and it has a typo:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "to": "https://social.example/users/evanp/followers",
  "object": {
    "type": "Note",
    "content": "Hello, Wrold!"
  }
}
```

We can use an `Update` activity to correct the mistake:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Update",
  "to": "https://social.example/users/evanp/followers",
  "object": {
    "id": "https://social.example/note/335",
    "content": "Hello, World!"
  }
}
```

Two points to note here: we need to include the `id` property, as returned from the `Create` activity, so that the ActivityPub API server knows which `Note` we're talking about. (There are probably a lot!) Second, we have to include only properties that we actually want to update; we can leave out `type`, for example.

Here's what the code would look like in Python:

```
# Create the note
activity = {
  "type": "Create",
  "object": {
    "type": "Note",
    "content": "Hello, Wrold!"
  }
}
result = do_activity(activity)
object_id = result['object']['id']

# Update it to correct the typo

do_activity({
  "type": "Update",
  "object": {
    "id": object_id,
    "content": "Hello, World!"
  }
})
```

We can add and remove properties with `Update` too. Let's say we want to remove the `content` property and replace it with `contentMap` in French and English:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Update",
  "to": "https://social.example/users/evanp/followers",
  "object": {
    "id": "https://social.example/note/335",
    "content": null,
    "contentMap": {
      "en": "Hello, World!",
      "fr": "Bonjour, le Monde !"
    }
  }
}
```

As you can see, to delete a property, you set its value to `null`. To add a property, you just include it in the update. In Python, the code would look something like this:

```
# Update again to use contentMap with en and fr
do_activity({
  "type": "Update",
  "object": {
    "id": object_id,
    "content": None,
    "contentMap": {
      "en": "Hello, World!",
      "fr": "Bonjour, le Monde!"
    }
  }
})
```

Note that the additive nature of the `Update` activity applies only to the properties of the `object`; it doesn't extend to properties of the object's properties. For example, this `Update` will result in a `contentMap` with only one language, not three:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Update",
  "to": "https://social.example/users/evanp/followers",
  "object": {
    "id": "https://social.example/note/335",
    "contentMap": {
      "es": "Buenas Días, El Mundo!"
    }
  }
}
```

In addition, you won't be able to update any server-managed properties like IDs, timestamps, or reaction collections:

```
# try to update the "published" property
do_activity({
  'type': 'Update',
  'object': {
    'id': object_id,
    'published': '1925-03-03T00:00:00Z'
  }
})
```

Delete

To complete the cycle, we have the `Delete` activity. This will delete the object in its `object` property:

```
do_activity({'type': 'Delete', 'object': object_id})
r = requests.get(object_id)
# This should return 404 or 410
print(r.status)
```

Objects that have been deleted will usually return a `404` or `410` status code when you attempt to fetch them, depending on the implementation. (`410` is the code for `Gone`, which is technically more correct, but some implementations use `404` instead to avoid giving away any information about the deleted object having existed.)

Sometimes deleted objects are kept in the server database, so that other objects that refer to them don't show errors. If so, the object type is usually changed to the special value `Tombstone`, and sometimes the `formerType` property is added. For example:

```
create_activity = do_activity({
  'type': 'Create',
  'object': {
    'type': 'Note',
```

```
    'content': 'Hello, World!'
  }
})
# Delete the object
do_activity({'type': 'Delete', 'object': create_activity['object']['id']})
# Get the create activity from the server
r = requests.get(create_activity['id'])
r.raise_for_status()
duplicate = r.json()
print(duplicate['object']['type']) # prints 'Tombstone'
print(duplicate['object']['formerType']) # prints 'Note'
```

Implicit Create

Finally, I want to call out a feature of the ActivityPub API called *implicit create*. I'm not a big fan of this feature, but some people really like it. It does exactly what it says: if you post an object to the outbox that's not an activity type, the API treats it like the object of a `Create` activity. This code will create a note:

```
create_activity = do_activity({'type': 'Note', 'content': 'Hello, World!'})
print(create_activity['type']) # prints 'Create'
print(create_activity['object']['type']) # prints 'Note'
```

This feature saves a little time and complexity, but it really depends on the ActivityPub API server to know which types are activities and which aren't. Differentiating them is not that hard for the main types used in ActivityPub or any of the other AS2-defined types. But what about extension types the server hasn't seen before?

```
create_activity = do_activity({
  '@context': [
    'https://www.w3.org/ns/activitystreams',
    'https://app.example/ns'],
  'type': 'Foo',
  'bar': 'baz'
})
```

Is the nonsense word `Foo` an activity type, and if so, should the system create the activity as shown? Or is it an object type, which would need to be wrapped in a `Create` activity? There's no easy way for the server to know. Some might guess; others might just refuse the activity entirely.

In general, if you use extension types with implicit create, you should use the multi-valued type feature we talked about in Chapter 2. This lets you say, "This object is a `Foo` and also an `Activity`," so the server will know what to do with it:

```
create_activity = do_activity({
  '@context': [
    'https://www.w3.org/ns/activitystreams',
    'https://app.example/ns'
  ],
  'type': ['Foo', 'Activity'],
```

```
  'bar': 'baz'
})
```

Alternately, you can use multivalued types for the implicit create feature, by passing input that is *not* an `Activity`—only an `Object`:

```
create_activity = do_activity({
  '@context': [
    'https://www.w3.org/ns/activitystreams',
    'https://app.example/ns'
  ],
  'type': ['Quux', 'Object'],
  'bar': 'kilroy'
})
```

Using multiple types for extensions to tie them back to their most similar AS2 vocabulary type, even if it's just `Activity` or `Object`, is a good general practice. It makes it easier for other Activity Streams processors to guess what to do with the extension object.

But, really, it doesn't make sense to push the envelope on implicit create just to save a few typing strokes. Implicit create can be useful for quick-and-dirty development, but providing your own activity types when you create new activities is more reliable.

Modifying the Social Graph

Of course, ActivityPub isn't just about creating content. We also need to manage the social connections between actors in the network, to control how content flows through that network.

Follow

As mentioned in "Reading Data: The Actor" on page 75, the two main feeds for tracking social connections in ActivityPub are the `followers` and `following` feeds of the actor. These have all the incoming and outgoing follow relationships, respectively.

To initiate this relationship, an actor uses the `Follow` activity type, with the other actor (the one being followed) in the `object` property slot:

```
follow_activity = do_activity({
  'type': 'Follow',
  'object': 'https://social.example/user/otheruser',
  'to': 'https://social.example/user/otheruser'})
```

Note that the other user needs to be included in the addressees of the activity. A good server will do this automatically, but it's better to just do it explicitly.

A `Follow` activity, in the ActivityPub world, doesn't always automatically result in changes to the `following` list. The followed actor can accept or reject the `Follow`

activity. This is good practice for personal accounts, especially if you're sharing personal information like family photos, personal observations, or physical location.

Accept and Reject

To accept the `Follow` activity, the followed user uses the `Accept` activity type, with the `Follow` activity as its object:

```
accept_activity = do_activity({
  'type': 'Accept',
  'object': 'https://social.example/user/otheruser/follow/10',
  'to': 'https://ap.example/user/firstuser'
})
```

If the user doesn't want to allow the `Follow`, they use a `Reject` activity:

```
reject_activity = do_activity({
  'type': 'Reject',
  'object': 'https://social.example/user/otheruser/follow/10'
})
```

Some ActivityPub servers won't send a `Reject` activity to the originating user, either local or remote. Although leaving the requester waiting for a reply might seem unfair, this is also a case of user safety. The rejected follower may react angrily to the rejection, so to protect the rejecting user, some developers choose to keep the follow request unresolved indefinitely.

Public accounts and bots often automate management of their social graphs and will send `Accept` activities instantaneously. Your app can tell the difference by using an extension property for actors, `manuallyApprovesFollowers`, which is a Boolean value (true or false). If it's true, delays in response might be due to the person on the other end just being busy.

Another useful extension, FEP-4ccd, adds `pendingFollowers` and `pendingFollowing` collections to the actor object. This gives the user a chance to review incoming follow activities. Note that, unlike the `followers` and `following` collections, the pending equivalents include the full `Follow` activity, so it's easy to `Accept` or `Reject` them. Here's a script that accepts all incoming followers:

```
actor = get_actor('https://social.example/user/otheruser/')
pendingFollowers = to_id(actor['pendingFollowers'])
for follow in items(pendingFollowers):
    do_activity({
      'type': 'Accept',
      'object': to_id(follow),
      'to': to_id(follow['actor'])
    })
```

Undo

Another useful activity type is `Undo`, which tries to undo the effects of the activity in its `object` property. That seems powerful, of course, but which activities it can actually undo depends on the server.

Most servers should be able to undo a `Like`, `Block`, `Follow`, or `Announce` activity. In particular, `Create`, `Add`, and `Remove` activities should use their opposite activity types instead of `Undo`.

For example, you can use `Undo` on each of the `Follow` activities in the actor's `pendingFollowing` collection to cancel any outstanding follows older than 30 days:

```
from datetime import datetime, timezone
actor = get_actor('https://ap.example/user/firstuser/')
if !('pendingFollowing' in actor):
  raise new Exception('pendingFollowing not supported on server')
pendingFollowing = to_id(actor['pendingFollowing'])
now = datetime.now(timezone.utc)
for follow in items(pendingFollowing):
    published = datetime.fromisoformat(follow['published'])
    if (now - published).days >= 30:
     do_activity({
       'type': 'Undo',
       'object': to_id(follow),
       'to': to_id(follow['object'])
```

Not all servers support the `pendingFollowers` or `pendingFollowing` properties, so it's a good idea to check before using them.

Undoing a `Delete` activity is helpful but not widely implemented. Many servers don't save the previous state of an object from just before it was deleted, so they can't correctly `Undo` the deletion. The next best thing is to re-create the content of the object, but that will have a new object `id`, and all `likes`, `shares`, and `replies` will be lost.

Managing Collections

Creating and modifying objects on the server are important, but organizing that content into collections is also important—for example, organizing a group of photos into an album or a group of actors into a contact list.

The `Collection` type in AS2 is the preferred way to make collections. In ActivityPub, we create collections the same way we create other types:

```
create_collection = do_activity({
    'type': 'Create',
    'object': {
        'type': 'Collection',
        'nameMap': { 'en': 'My vacation photos' }
```

```
    }
})
collection_id = create_collection['object']['id']
```

Add

After creating a collection, you can add items to it with the `Add` activity type:

```
do_activity({
   'type': 'Add',
   'object': 'https://photos.example/evanp/photo/1',
   'target': collection_id
})
```

Note that the thing being put into the collection is the `object`, and the collection it is being put into is the `target`.

The preceding code doesn't specify exactly *where* in the collection the object will be put. However, it's a common pattern to sort ActivityPub collections in reverse order of how the objects were added. So the most recently added items will be the first few items, and the oldest ones will be the last few items. Following this pattern isn't required, but it's common enough that that would be a good place to look.

It may be tempting to modify the contents of a `Collection` in a different way—say, by using the `Update` activity to change its `items` property, or by using the `Add` activity on one of its `CollectionPage` pages. Although this may work with some server implementations, it's not a good practice. The collection interface is flexible, but servers may implement streams and collections in different ways under the hood. Sticking with the official interface and using `Add` is the best way to get good interoperability.

Remove

To take an object out of a collection, use the `Remove` activity type:

```
do_activity({
   'type': 'Remove',
   'object': 'https://photos.example/evanp/photo/1',
   'target': collection_id
})
```

Here, you can see the benefits of not specifying an order. You're not responsible for finding the object or patching up the collection, as you would be with an array in a common programming language. The ActivityPub server does all that directly.

Note that the collection is in the `target`, even though, at the end of the process, the object will not be in the collection anymore. (In the trade-off between parallelism of `Add` and `Remove` and using precise terms, the creators of ActivityPub chose parallelism.)

Finally, removing an object from a collection does not delete the object. If you want it gone, use a `Delete` activity. It's best to do the `Remove` activity first, then the `Delete`; if you do it the other way around, the deleted object may still be in the collection with a `Tombstone` type. That's what tombstones are for, after all—holding the place of deleted objects in collections.

Update

It's possible to use the other content-management activities to modify other aspects of collections. For example, to update a title, use `Update`:

```
do_activity({
    'type': 'Update',
    'object': {
        'id': collection_id,
        'nameMap': { 'en': 'My AWESOME vacation photos' }
    }
})
```

Delete

To get rid of a collection, use `Delete`:

```
do_activity({
    'type': 'Delete',
    'object': collection_id
})
```

Note that deleting a collection doesn't delete all the objects in the collection. If you want to do that, you should first delete each item, then delete the collection. If you do it in the other order, finding which items were in the collection will be difficult:

```
for item in items(collection):
   do_activity({'type': 'Delete', 'object': to_id(item)})
do_activity({'type': 'Delete', 'object': collection_id})
```

Reacting

A key part of the social-network experience is reacting to existing content. That's what makes social networks interactive; we can scan our `inbox` for the latest content our friends and family have posted, and when we see something interesting, we can give them feedback about it.

Like

The most basic reaction is the `Like` activity, which tells the creator (and everyone else on the fediverse who can see it) that you like what they made. The `object` of the activity is the thing that you liked. It's possible to `Like` other objects on the fediverse

besides content objects (text, images, video, audio). For example, you can like an actor, or an activity, or a collection:

```
like_activity = do_activity({
  'type': 'Like',
  'object': 'https://photo.example/evanp/photo/13',
  'to': ['as:Public', 'https://photo.example/evanp']
})
```

Liking an object has a couple of important side effects, which the ActivityPub server will take care of. First, it adds the liked object to the actor's `liked` collection, if one exists. This collection is a reverse-chronological list of all the activity objects that the actor has liked:

```
id = `https://photo.example/evanp/photo/13`
actor = get_logged_in_actor()
for item in items(actor['liked']):
  if to_id(item) == id:
    print('Found it!')
```

Second, the ActivityPub server adds the `Like` activity to the `likes` collection on the object. Again, this is a reverse-chronological collection of all `Like` activities with this object as the `object`:

```
id = `https://photo.example/evanp/photo/13`
photo = get_object(id)
for activity in items(photo['likes']):
  if to_id(activity) == like_activity['id']:
    print('Found it!')
```

If a user likes an object accidentally or changes their mind over time, the `Undo` activity type can be used to undo the `Like` activity:

```
do_activity({'type': 'Undo', 'object': to_id(like_activity)})
```

This will have two side effects: removing the object from the actor's `liked` collection, and removing the `Like` activity from the object's `likes` collection.

Announce

Another important reaction to posted content is *sharing*. This distributes content we think is important or insightful to a wider audience. And each person in that audience may share it in turn. Cascading shares of an activity object, where people in that object's audience share it in turn, can give it broader and broader range. Some images and text posted on the fediverse have been shared tens of thousands of times.

To share an object in the ActivityPub API, use the `Announce` activity type:

```
do_activity({'type': 'Announce',
             'object':`https://photo.example/evanp/photo/13`,
             'to': ['as:Public', 'https://photo.example/evanp']})
```

The `Announce` activity is most effective when the original object is available to the public and the announcer shares to the public again. Sharing an object to a smaller group or even to selected actors is also possible. Sharing to a larger group won't work as well as it might seem, however. If the object's author distributed it to their own followers, sharing it will not make it available to more people. Only the author's followers will be able to access it.

As a side effect, the ActivityPub server will add the `Announce` activity to the object's `shares` collection. (No collection of an actor's shared objects corresponds to the `liked` collection.) This lets the original creator count the shares and see where the original content has gone.

You can announce any kind of activity object—an activity, an actor, whatever. But some ActivityPub processors won't know what to do with a shared object that's not more or less a digital content type, such as `Note`, `Article`, `Image`, `Video`, or `Document`.

inReplyTo

Finally, an important way to engage with other people's content is to reply or comment on it—telling them it's great, answering a question, making a joke. ActivityPub doesn't have a specific type for comments. Instead, we use a special property, `inReplyTo`, on a content object to say that it's "in reply to" another content object:

```
create_reply = do_activity({'type': 'Create',
  'to': ['https://photos.example/evanp', 'https://photos.example/evanp/followers'],
'object': {
   'type': 'Note',
   'content': 'This photo looks great!',
   'inReplyTo': 'https://photos.example/evanp/photo/664'
}})
```

Any type can have an `inReplyTo` property, but the most common and best supported type is `Note`. The structure of a `Note`, as a short text of about one paragraph, is essentially what people think a "comment" should be. Also, any type can have replies (an activity or an actor, say) but as with other types of reactions, the most likely types to be handled well are content types—text, images, videos, audio, and other documents.

When you create an object with `inReplyTo`, the ActivityPub server adds it to the `replies` collection of the replied-to object. This makes it easy to see what other people have said about the original work. The `object` is what's added to the collection, not the activity. To continue the preceding example:

```
photo_object = get_object(create_reply['object']['inReplyTo'])
for reply in items(photo_object['replies']):
  if reply['id'] == create_reply['object']['id']:
    print('Found it!')
```

Deleting a reply may remove it from the original's `replies` collection, but it's more likely to leave a `Tombstone` object in the collection.

What happens when someone else replies to the reply? The basic answer is that the second reply ends up in the `replies` collection of the first reply, but not in the replies to the original post. A tree structure builds up among the replies.

Managing this tree is one of the more difficult aspects of developing a general-purpose ActivityPub client, such as a mobile client or a web client. Typically, given any particular content, like a `Note`, such a user interface should show all the *ancestors* of the object—the object that it's in reply to, as well as any object that it's in turn in reply to, and so on until the original posted object is reached:

```
def ancestors(obj):
    if 'inReplyTo' not in obj:
        return []
    else:
        parent = get_object(obj['inReplyTo'])
        return [parent].concat(ancestors(parent))
```

The user interface should also show all the object's *descendants*—the members of the object's `replies` collection, plus all the replies to *those* objects, and so on until the leaf nodes of the tree have empty `replies` collections. Here's an example function in Python that builds a tree. Each node, including the object itself and all its replies, is a *tuple*—a Python data structure representing a fixed number of values:

```
def make_node(obj):
    node = [get_object(obj), []]
    for item in items(obj['replies']):
        node[1].append(make_node(item))
    return node

def descendants(obj):
    return make_node(obj)
```

Of course, online conversations can stretch to thousands or tens of thousands of comments and replies. Another common representation is to show the first *N* direct replies (say, 5 or 10) with a way to show more (like a More button), and use another mechanism for showing further replies to each, if they exist.

With so much data requiring so many requests, reply trees are a great candidate for caching optimization.

Ensuring User Safety

Social networks include lots of kinds of people, and unfortunately they don't always get along. Some people intentionally aggravate, harass, or intimidate users on the network; others try to scam or spam large groups; others are annoying or irritating unintentionally. Sometimes we just need a break from people we know and care about.

Block

For whatever reason, it's important that people on social networks can control who they interact with. The `Block` activity in ActivityPub blocks another actor on the network from interacting with the current user:

```
block_activity = do_activity({
  "type": "Block",
  "object": "https://social.example/users/mallory"
})
```

Blocking an actor has several side effects. Once an actor has been blocked, the ActivityPub server will not let them read or react to the user's created objects. Most ActivityPub implementations also cancel any follow relationship between the two actors, disallow new follows, and filter the blocked actor's activities and content from the blocking actor's streams.

However, as you may have noticed, social networking is a complex domain. It's hard to balance all users' expectations, particularly with blocking: some users expect the blocked user to be completely erased from their experience, whereas others just want to end the existing relationship with the blocked user and prevent any new ones.

Blocking someone is a sensitive activity. If the blocked user is particularly hostile, they can take it badly and try to retaliate. For this reason, most implementations don't share blocking information with the blocked user. Instead, the blocked user's client software will get errors when they try to fetch the blocking actor's content or interact with them.

It's possible to `Undo` a `Block` activity:

```
do_activity({
  "type": "Undo",
  "object": block_activity['id']
})
```

Again, ActivityPub servers undo the `Block` activity to a greater or lesser extent. Usually, undoing a `Block` activity makes new interactions possible and does not filter the (formerly) blocked user's content or activities (even ones created during the period of the block). But undoing a block does not usually reinstate any previously existing follow connections.

There's no standard collection of all actors the current actor has blocked. However, an extension property, `blocked`, defined in FEP c648, returns all of that actor's `Block` activities. It's available only to that actor, to prevent abuse by blocked users:

```
if 'blocked' in actor:
    for block_activity in items(actor['blocked']):
        print(block_activity['object']['name'])
```

As with `pendingFollowing`, this code checks for the presence of the `blocked` property before using it.

Flag

Another important user-safety feature is the `Flag` activity. This lets a user report unacceptable content or problematic profiles to trust and safety teams for review:

```
do_activity({
   "type": "Flag",
   "contentMap": {
     "en": "This note uses an unacceptable slur."
   },
   "object": {
      "id": "https://social.example/users/tim/note/177"
      "type": "Note"
   }
})
```

A `Flag` activity can be used for content objects like `Note` and `Image` objects, for `Activity` objects, and for actors, as in this example:

```
do_activity({
   "type": "Flag",
   "object": {
        "id": "https://social.example/users/tim",
        "type": "Person"
   }
})
```

The activity doesn't have any specified addressees; the ActivityPub server is responsible for getting the report handled somehow. Human moderators can use this kind of report to disable user accounts or limit their abilities to interact with others. A `Flag` activity is also helpful in training algorithms to detect unacceptable behavior, like spam or abuse.

Uploading Files

Short HTML5 texts like `Note` objects, and even longer texts like `Article` objects, are usually held right in the body of their AS2 JSON representations. But we use lots of types of media to communicate in social networks—photos, GIFs, videos, audio recordings, binary documents, and so on. Each has a complicated standard file format.

One possibility for sharing files in the ActivityPub API is to upload the files "somewhere" (for instance, the Internet Archive's wonderful public data-sharing service) and then include their URLs in the AS2 representation of the object. Here's an example:

```
file_url = (
  "https://archive.org/download/evanp_rubikscube_1/"
  + "VID_20180714_115633479_HDR.ogv"
)

create_activity = do_activity({
  "type": "Create",
  "to": [current_actor["followers"]],
  "object": {
    "type": "Video",
    "nameMap": {"en": "Me solving the Rubik's Cube"},
    "url": {"type": "Link", "mediaType": "video/ogg", "href": file_url},
  },
})
```

Note that this is a `Create` activity. The Internet Archive doesn't provide ActivityPub IDs for uploaded files, so we need to ask the ActivityPub API server to create a JSON-LD representation of the file and give it an ID. And, yes, some conceptual slippage occurs here between the JSON representation and the binary file.

Another problem is that the Internet Archive won't enforce any access or authorization policies on the object (see "Understanding the Authorization Model" on page 111). If an object is shared only to the current user's followers—not the general public—the Internet Archive doesn't know that. It will happily share the file with anyone who has the URL.[1] Controlling access to the JSON-LD representation of the file can prevent that URL from spreading too far, but this kind of "security through obscurity" isn't very secure. It's especially unhelpful if the creator needs to modify access after the initial post—say, by blocking an uncooperative follower or changing the post's addressees (that is, the actors named in its `to`, `cc`, `bto`, `bcc`, and `audience` attributes). Once that URL is shared, anyone can use it.

1 This isn't a criticism of the Internet Archive, which is specifically for publicly sharing data. It's just a weakness of this kind of file sharing with ActivityPub.

The other option is using ActivityPub's file-uploading feature. This was a difficult topic in standardizing ActivityPub, and a final recommendation wasn't included in the spec. The file upload section of ActivityPub continues to be worked on by the W3C Social Web Incubator Community Group (SocialCG) and may be part of a future specification. As I write this in 2024, some ActivityPub API implementations are using the uploading process defined here.

To upload a file, use the `uploadMedia` endpoint, defined in the `endpoints` collection in the actor model:

```
current_actor = get_current_actor()
if ('endpoints' in current_actor and 'uploadMedia' in current_actor['endpoints']):
  upload_endpoint = current_actor['endpoints']['uploadMedia']
else:
  upload_endpoint = None
```

The upload media endpoint accepts an HTTP POST request with a specific payload. It's a *multipart/form-data payload*, which is how most browsers will upload files to a server. The payload has two parts (that's the *multipart*): a binary file named *file* and a JSON-LD descriptor of that file, named *object*. Here's a simple example in Python:

```
filename = 'test.png'
uploadMedia = actor['endpoints']['uploadMedia']

descriptor = {
  'to': [actor['followers']],
  'type': 'Image',
  'nameMap': {'en': 'Test image file'},
  'summaryMap': {'en': 'A simple purple five-pointed star on a white background'}
}

multipart_form_data = {
  'file': (
    filename,
    open(filename, 'rb'),
    'image/png'
  ),
  'object': (
    'descriptor.jsonld',
    json.dumps(descriptor),
    'application/activity+json'
  )
}

oauth = get_session()
r = oauth.post(uploadMedia, files=multipart_form_data)
r.raise_for_status()
create_activity_url = r.headers.get('Location')
print(create_activity_url)
```

Note that the image is posted only to the actor's `followers` collection. The server will make sure that only followers can view the file, as the actor intends.

The server makes a new `Create` activity, with the new `Image` as the `object` and the URL of the uploaded file as the image's `url` property. The `Location` header returned by the upload media endpoint will have the correct URL for the `Create` activity, which can be used as you follow your nose to find the `Image` object and the uploaded file's URL:

```
r = oauth.get(create_activity_url)
r.raise_for_status()
create_activity = r.json()
image_id = create_activity['object']['id']
image_url = create_activity['object']['url']
```

You can then use the image ID for other things, like adding the image to a collection:

```
do_activity({
    'type': 'Add',
    'object': image_id,
    'target': collection_id
})
```

You can use the ID as an attachment:

```
do_activity({
    'type': 'Create',
    'object': {
        'type': 'Note',
        'contentMap': {'en': 'Check out this test image I uploaded!'},
        'attachment': [image_id]
    }
})
```

You can even set it as your avatar photo:

```
actor = get_current_actor()
do_activity({
    'type': 'Update',
    'object': {
        'id': actor['id'],
        'icon': image_id
    }
})
```

Note that all the uses of the uploaded file are for `image_id`, *not* the binary file URL. This matches our regular usage of JSON-LD representations of objects in ActivityPub.

Although this file upload API is powerful, it has weaknesses. First, there's no standard way to update or replace an uploaded binary file (for example, correcting an image's size or cropping its margins). You can get a lot of value out of uploading a different

file and replacing the uses of your first file with the second file (say, by updating the `icon` property of your profile). However, the URL of the first file will keep returning your first upload.

Second, there's no standard way to delete an uploaded file! You can try an activity like this:

```
do_activity({
    'type': 'Delete',
    'object': image_id
    }
})
```

However, whether the binary stream will be deleted depends on your implementation.

Finally, this API is very chatty, especially if you're uploading a lot of files. It generates one `Create` activity per uploaded file, which can be a lot for your social stream if you're uploading your entire slideshow from your trip to the Grand Canyon last summer.

It's actually possible to give a different structure in the JSON descriptor. If you use an `Activity` type, the upload handler will use your JSON as the skeleton of the new activity rather than using a `Create` activity. So you can have a descriptor like this instead:

```
descriptor = {
  'to': [actor['followers']],
  'type': 'Add',
  'object': {
     'type': 'Image',
     'nameMap': {'en': 'Test image file'},
     'summaryMap': {'en': 'A purple five-pointed star on a white background'}
   },
   'target': collection_id
}
```

As ActivityPub matures, we can expect more exotic permutations of uploads, such as uploading multiple files in one activity. For now, the power of an upload endpoint that enforces the ActivityPub authorization model is still pretty cool.

Understanding the Authorization Model

This brings me to topic of the ActivityPub authorization model. After you've logged in with OAuth 2.0, what can you do with different kinds of objects? The ActivityPub specification doesn't provide a well-defined model, but many implementations follow simple, intuitive rules for allowing different users access to objects. These rules reflect what users often consider acceptable behavior in social-network applications. They include the following:

- As the starting point for ActivityPub API actions, actor objects need to be readable for anyone on the fediverse, including unauthenticated users. They need to be publicly available by default.
- The creator of an object—more specifically, the actor who did the `Create` activity that created the object—can read, update, and delete the object.
- The addressees of an object can read the object, but cannot modify or delete it.
- If the object is addressed to the special addressee `as:Public`, anyone can read the object—even unauthenticated users.
- If an object's addressees include collections of actors (such as lists of users, or special lists like `followers` collections), members of those collections can read the object. If an actor leaves the collection (say, by unfollowing the creator) or is removed by the collection's owner, they lose read access to the object.
- Actors who can read an object can also react by liking, replying, or sharing it.
- Actors blocked by someone cannot read or react to objects created by the blocking user (even those objects that are available to unauthenticated users).
- Actors can read all their own collections.
- Other actors may read some of an actor's collections (typically, `outbox`, `following`, `followers`, and `liked`).
- ActivityPub servers will filter collections to include only items that the authenticated user can read: for example, reading an actor's `outbox` with such a filter would return only the items shared with the public or with the current user. Implementations sometimes change the `totalItems` property of a collection that has been filtered this way, but sometimes they don't. So treat `totalItems` as a maximum value, not an exact one!
- Authorization for the JSON-LD representation of a media file (like an image, video, audio file, or document) is identical to that of the media file itself. So if an actor can read the JSON-LD, they can probably also read the binary file.

These aren't hard-and-fast rules, and you should write your ActivityPub API client software to tolerate variations. In particular, some implementations allow only authenticated users to access objects by ID, even if the object is public.

These authorization rules are enforced only when you request an object using the HTTPS URL in the `id` property. Once you have that data, you can pass it around, store it insecurely, whatever. Well-behaved ActivityPub API clients follow these authorization guidelines when handling people's personal social data. Badly behaved API clients (and their users) find themselves blocked by servers. Don't be That Client!

Optimizing the ActivityPub API

The ActivityPub API is pretty efficient, but a few simple enhancements can improve your clients' performance. This section offers some tips.

Use an HTTP Cache

Clients will fetch the same object by ID many times during the lifetime of the program. Using an HTTP caching library can substantially reduce the number of HTTP requests you make. The requests-cache library, for example, can seamlessly add HTTP caching to any requests-using app. Using a cache has downsides, of course; if your app is multiuser, you need to make sure you don't leak private data between users by sharing the same cache. If the ActivityPub server supports the `Vary` header, this will usually give a clue to caching libraries of how to keep the cached data separate.

Use Data You Already Have

When you fetch an `Image` object, its `attributedTo` property is sometimes simply the ID of the image's creator. But the `attributedTo` property can also be an object.

As a first pass, you might write your code to throw away everything that's not an `id` in an attached object:

```
if isinstance(image['attributedTo'], dict):
  id = image['attributedTo']['id']
else:
  id = image['attributedTo']

r = oauth.get(id)
r.raise_for_status()
results = r.json()
author_name = results['name']
```

However, the `name` property may already be in the object property `attributedTo`:

```
author = image['attributedTo']
if isinstance(author, dict) and 'name' in author:
   author_name = author['name']
else:
   id = author['id'] if isinstance(author, dict) else author
   # … get full author record from here
```

It pays to check for what you need before spending a full HTTP round trip (including parsing the output JSON) getting data you already have!

Use Low-Resolution Representations

If you need additional data that's not provided in the brief representation of an object's property, use what you have with simple placeholders and request the full representation in the background. When the full representation comes in, replace the low-res version with the full version. This rule is just a refinement of the rule about using data you already have.

Although this technique isn't technically faster, it sure *feels* faster to the user, who gets a responsive UI almost immediately, with a little update a moment later. With our previous `attributedTo` example, we can show a placeholder avatar while the full data is being downloaded:

```
# an example method for showing author info
author = image['attributedTo']
set_author_info(name=author['name'],avatar=placeholder)
id = author['id']
r = oauth.get(id)
r.raise_for_status()
full = r.json()

# set the info again once it has been received
set_author_info(name=full['name'], avatar=full['icon']['url'])
```

Reuse the Output of Posted Activities

If the ActivityPub server returns 201 results as `application/activity+json` payloads, you should definitely use that output instead of trying to discover where your activity went.

Use GZIP Compression

If your HTTP client library supports it, you can shorten your download time by using GZIP compression. The requests library, for example, will expand GZIPped content by default, so just saying you'll accept GZIPped payloads helps out:

```
headers = {'Accept-Encoding': 'gzip'}
oauth.get(image_id, headers=headers)
```

Use Keep-Alive Connections

The time it takes to look up a Domain Name System (DNS) name, establish a TCP connection, and negotiate SSL parameters is a significant chunk of the time needed to fill most HTTP requests. *Keep-alive connections* refers to a technique in which the connection between client and server is kept open after a response is received, in case another request is coming soon. Usually, these connections stay up for tens of seconds, sometimes even minutes. With an active social-network client, you'll likely make a lot of API calls during that keep-alive time.

Different programming languages and libraries support keep-alive with different techniques. For the requests library, for instance, the `session` abstraction hides a useful connection-pooling mechanism that includes keep-alive connections. Other libraries use a similar abstraction, so looking for *session* or *connection* mechanisms will probably put you on the right track.

Understanding What's Missing

The ActivityPub API isn't perfect, not by a long shot. This section briefly addresses some of the issues you might run into while using it:

Collections are sorted, searched, and filtered on the client side
: This is one of the harder parts of using the ActivityPub API. The `inbox` collection, for example, shows every single activity in reverse-chronological order. If you want to show, say, only activities that mention the word *toast,* or you want to cluster activities by the author's last name, you would need to download the entire collection, page by page, and sort or filter it on the client side. (Extensions are in the works for pushing this sorting and filtering to the server side, but they're still in the standardization phase as of this writing.)

The actor is the primary entry point
: Everything we've talked about here focuses on the actor: getting the actor's inbox, adding to the actor's followers, and so on. But what about server-wide searches? What if you want to find everyone on the server whose name is Jeff? Or every image posted in the last 24 hours related to houseplants?

  Serverwide data is not well covered in ActivityPub. You can use an optional endpoint, called `sharedInbox`, to follow all the public activities by any actor on the server. But you need an actor ID to find that endpoint! Some software, like Mastodon, provides an actor that represents the entire server, which is a good way to manage this issue.

Users have limited access to the collections they create
: As a user, there's no easy way in ActivityPub to find, say, all the photo albums you've created or all the contact lists you've used. The actor object has an optional `streams` property, but it's only loosely defined. At worst, if you want to find "all your stuff," you'd have to sort through the entire `outbox` stream to find `Create` activities for each type. That's a lot of work!

There is no administrator interface
: Some ActivityPub API servers give certain users more capabilities—like updating or deleting other users' content, approving or denying user registration, or updating the server actor. But the specification doesn't clearly define how to grant those capabilities, what they would be, and how users can apply them.

RESTful principles aren't followed throughout
: One of REST's core tenets is to use HTTP methods to modify resources. But instead of using UPDATE or PATCH on an ActivityPub object's `id` to change the object, we instead post an `Update` activity to the actor's outbox. Instead of using the DELETE HTTP verb, we use a `Delete` activity. Other actions like following or blocking might also map to adding an object to a collection with a POST.

There's no push mechanism
: To get the new items that arrive in the `inbox`, the client needs to GET the `inbox`. And to see if there are any other new items, the client needs to GET the `inbox` again. No standard mechanism exists to push new items to the client; the client has to keep polling. Some client implementations use tools like WebSocket (RFC 6455) or server-sent events to update clients, but there's not yet a standard for pushing ActivityPub API feeds to the client.

Fortunately, ActivityPub is an extensible API, and its developer community is working to address these problems. Let's hope they get easier with time!

Conclusion

The ActivityPub API provides a simple, extensible way to create client software for social-network services. It's based on an extensive vocabulary of social-network types and properties: the Activity Vocabulary. The API adds simple rules for resolvable identities that makes a readable network of objects connected through the web. And on top of that is layered a RESTful mechanism for executing and distributing a dozen or so well-defined activity types.

In Chapter 5, we'll discuss how to implement extensions to the ActivityPub API that give the potential for infinitely more types of social interaction. But first, we're going to take a deep dive into the ActivityPub federation protocol to show how servers can receive and deliver activities from entirely remote social networks.

CHAPTER 4
The ActivityPub Protocol

The standards we've talked about so far—Activity Streams 2.0 and the ActivityPub API—are important because they standardized something that already existed in most social networks: data structures and client-to-server APIs. That's a big achievement.

But the part of ActivityPub that changed the way people communicate on the internet—the part that many groups had tried before, without success—was the federation protocol. The *federation protocol* is what lets people on different services talk to one another as if they're all on the same service. It enables the backend connection that delivers activities by users on one server to users on another server.

In this chapter, I'll present the architecture of the ActivityPub federation. I'll show you where the ActivityPub API leaves off and the ActivityPub federation protocol starts up. I'll describe the authentication framework that lets people and servers on the fediverse identify themselves to one another. Then, I'll talk about how activities get delivered from one server to another. I'll go over the main types of activities we see on the fediverse and what well-behaved servers should do about each of them. Finally, I'll cover how to connect conversations and threads on the fediverse.

Exploring an Extended Example

To illustrate the topics in this chapter, I'll use an extended example of an ActivityPub protocol implementation. Since the protocol is for a social-network federation, I was tempted to create a small social-network server, complete with user account management and an implementation of the ActivityPub API on the frontend. To keep the example tight and to keep this chapter short, without a lot of extraneous code, however, I decided instead to develop a *bot server*.

Bots are automated fediverse users; instead of real human beings posting content and creating activities, their simple code lets them perform actions that humans might do. Bots often connect to an ActivityPub server by using the ActivityPub API or another API, but for this example I'm going to build a plug-in interface so that the bots run on the server, inside the server process. The server will be usable only for bots.

The *bot interface* implemented in the service will let the bots send notes, follow and block other users, and like and share content. The bots can also react to activities from other users on the fediverse. Implementing the bot interface will let me show you how best to implement these features in your ActivityPub server, without getting bogged down in what the affordances are for doing those tasks. Figure 4-1 shows the architecture for *activitypub.bot*. Its ActivityPub protocol handler will perform default processing for some types of activities, but others are passed through to plug-in software (the bots), which have a library of services they can use to enact their behavior.

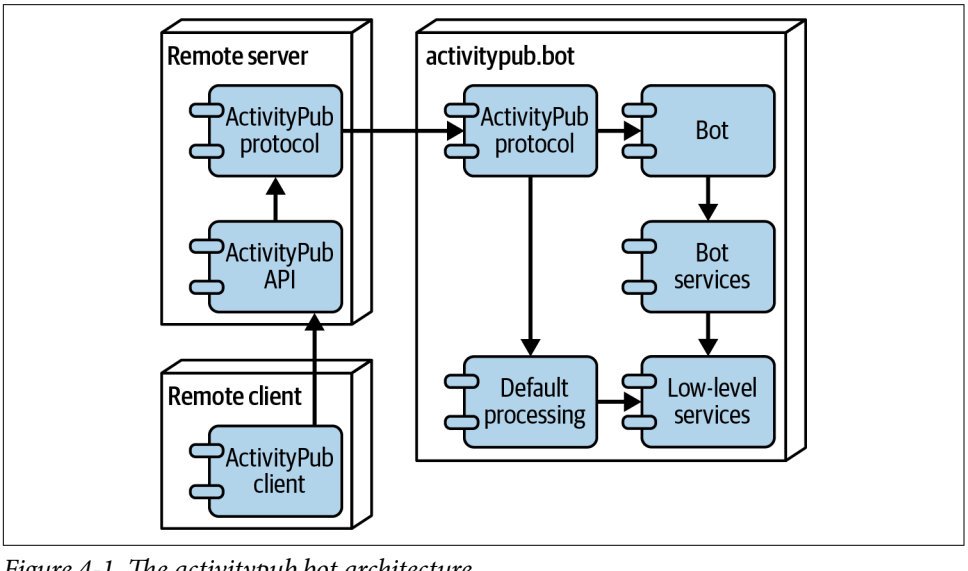

Figure 4-1. The activitypub.bot architecture

The project is called *activitypub.bot* because that's the best domain I was able to get. I'll show you a few sample bots in this chapter; others may be implemented by the time you read this book. You can see the project code on GitHub, along with documentation on how to interact with the bots.

The code for this extended example is in JavaScript—as of this writing in 2024, the most popular programming language on the planet for more than a decade. On the off chance that you've never heard of JavaScript, it is a dynamically typed, object-oriented programming language reminiscent of C, C++, and Java, with a little hint of functional languages like Lisp or Scheme mixed in. It has been the main scripting language for web pages since the mid-2000s, and with the development of Node.js has become popular for server-side web development also.

If you want to learn JavaScript or need a refresher, *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly 2020) is a comprehensive learning and reference guide. Also, Douglas Crockford's *JavaScript: The Good Parts* (O'Reilly, 2008) remains a notable and staunchly opinionated book that makes up for its somewhat outdated syntax recommendations with a joyfully irascible tone. It's the book that made me fall in love with the language. Finally, *Web Development with Node and Express* by Ethan Brown (O'Reilly, 2019) gets specific about building web servers with Node.js, including Express.js, the web server framework I used for this example.

To manage AS2 parsing and formatting, I use James Snell's excellent *activitystrea.ms* library from NPM, the Node.js package repository. James is my coeditor on the AS2 specification and definitely the creative genius behind the whole endeavor. His AS2 package is fun and easy to use. Here's how to import it:

```
const as2import = util.promisify(as2.import)
const foo = await as2import({'name': 'Evan', 'type': 'Person'})
console.log(foo.name)
```

Understanding the Shape of Federated Social Networking

Actors on a social network publish and receive activity data. Yes, I know, that's a *really* boring way to describe the fun, emotion, and interactivity of social-network software—but hopefully it will help visualize the shape of the network.

For many social-network systems, users have a client that they use to interact with a remote server. Each user has their own client; activities go from the client to the server, and then out to other users' clients.

Figure 4-2 shows a model of two users on the same server. The client software on user 1's device creates an activity by posting to the user's outbox, which creates a representation in the internal data storage. When user 2 goes to read their home page in their own API client, they call the inbox GET handler, which retrieves activities from data storage, including in this case the activity by user 1.

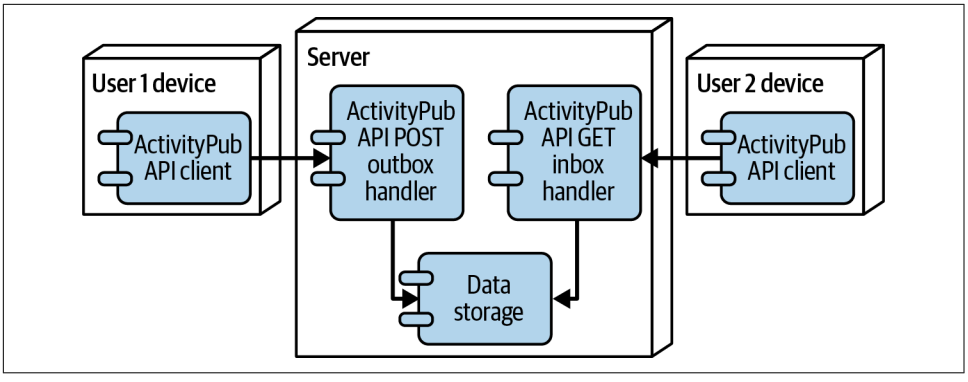

Figure 4-2. Two users on the same server using the ActivityPub API

One feature that makes the ActivityPub network special is that users don't have to have accounts on the same server. Figure 4-3 shows what it looks like when the users are on separate servers.

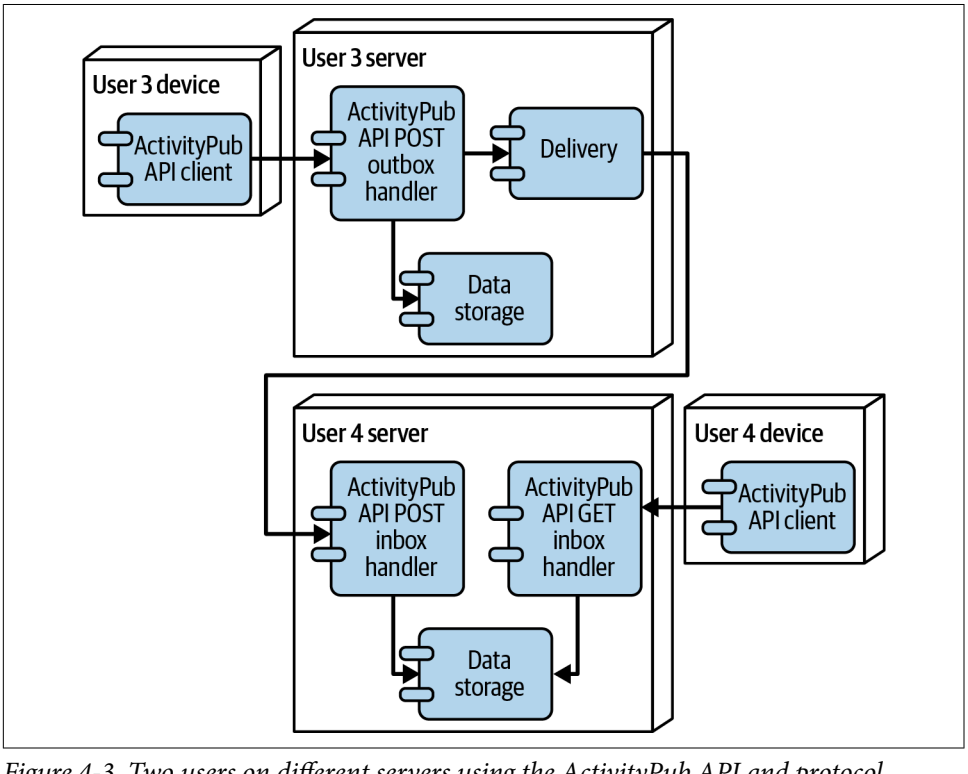

Figure 4-3. Two users on different servers using the ActivityPub API and protocol

This is a different layout: user 3 and user 4 have their accounts on different servers. When user 3 POSTs a new activity to their outbox, it still goes into local data storage, but it also gets sent to a delivery component. The delivery component uses the ActivityPub *protocol* to POST the activity to the inbox of each and every user addressed in the activity. Those inbox-handling components store the activity in the remote server data storage, so that when user 4 loads their inbox feed, the new activity will be there.

This is a simplified model. A lot more chatter occurs between servers than just delivering activities. But it's important to understand that ActivityPub servers are relatively independent. They can use different software implementations, run in different server environments, and have different domain names. They don't have shared storage; all the interactions are via REST method calls, whether reading with GET or writing with POST.

So, how does the ActivityPub protocol differ from the ActivityPub API? As you'll see in the next section, not much at all.

An API Becomes a Protocol

As you saw in Chapter 3, the ActivityPub API—the part of the ActivityPub specification concerned with letting client applications connect to servers—has two main parts:

A read-only structure
: The read-only structure is based on the `id` properties of ActivityPub objects. You can read the information about an ActivityPub object by fetching the URL that is its `id`, and you can fetch related objects (its creator, its parts, and so on) by fetching the URLs in various properties of the main object. So, to get the creator of a `Note` object, you get the URL from its `attributedTo` property. This is the "follow your nose" part of the API; you start with an `id` URL, and then use the properties to follow the path to the parts you're looking for.

A write-only part
: Here, the client uses an HTTP POST request to send an AS2 activity object to the actor's `outbox` URL. Each activity that is posted to the outbox can have side effects; for example, a `Like` activity will cause its `object` property to be added to the actor's `liked` collection.

So, an ActivityPub API server has the responsibilities of making objects available for reading, accepting activities and realizing their side effects, and running authentication and authorization checks to make sure only the right actors can read and write data.

The ActivityPub federation protocol is extremely similar to the API in its structure. It also has a read-only and a write-only section. The *read-only* structure is exactly the

same, providing access for other servers to read AS2 objects and follow their properties to other, related AS2 objects. Once you've implemented the read-only structure of the ActivityPub API, extending it to work for the federation protocol too is easy; the main difference is the authentication mechanism that's used.

The *write-only* structure works almost the same way. But while the actor's `outbox` contains all the activities that they performed, the actor's `inbox` contains all the activities that others have performed and sent to the actor. To send an activity to an actor, an originating server POSTs the AS2 representation of the activity to the receiving actor's `inbox` URL.

To support the ActivityPub federation protocol, then, a server has to do the following:

- Implement read-only access to its AS2 objects
- Deliver any activities that its actors create to the inboxes of addressed actors on other networks
- Accept incoming activities that actors on other networks send to its own users' inboxes
- Realize the side effects those remote activities might have
- Implement the authentication and authorization checks needed to preserve its actors' privacy

This parallel between the API and the federation protocol is completely on purpose. When the SocialWG was developing ActivityPub, we wanted it to be easy to implement one part of the standard if you were already working on the other part. So, the read-only parts are more or less the same, and the write-only parts are extremely parallel—just the difference between the `outbox` and the `inbox`, as well as the different side effects for locally created activities versus remote activities.

The biggest difference between the API and the federation protocol is authentication. The OAuth 2.0 mechanism used for the API depends on the actor having an account on the API server. Since remote actors don't have accounts on the local server (that's what makes them remote!) we need a different way to identify HTTP GET and POST requests made on behalf of those remote users. The most popular mechanism for this on the fediverse today is a standard called HTTP Signatures.

Using HTTP Signatures

HTTP Signatures is a standard developed as a request for comment (RFC) at the Internet Engineering Task Force (IETF). The IETF is a standards body, like the W3C, that brings together experts to define the rules that make the internet work. Despite the noncommittal name (request for comment), every RFC represents carefully considered rules for internet protocols and software behavior.

That's the good news: great engineering leadership by tried-and-true experts, working to keep our internet safe. The *bad* news is that, when software projects like Mastodon were first implementing ActivityPub in the late 2010s, the HTTP Signatures RFC was still under development. The version of the RFC that was implemented is incompatible with the latest drafts and technically expired in 2019. Because it's widely implemented, a lot of inertia is holding people back from changing to a newer version.

But wait! There's even more bad news. Even within the parameters of the draft HTTP Signatures RFC, a lot of possibilities remain for storing and sharing keys as well as ways for making signatures. But most fediverse software supports only one, or at most a few, of those choices. And sometimes the only way you'll know that your choices were wrong is that no other fediverse software can talk to yours.

Now that you're thoroughly discouraged, let's pull back and look at the big picture. As of this writing, the W3C SocialCG has released a great new report on using HTTP Signatures. It brings this component of the social web stack out of the realm of folklore and into the realm of agreed-upon standards. With luck, this stabilization will engender more tools and libraries that support HTTP Signatures.

Hundreds of independent implementations of HTTP Signatures with ActivityPub exist today. Tens of millions of ActivityPub activities are delivered via the federation protocol on a daily basis. Sure, the authorization mechanism is a little tricky, but once you've got it working, this foundational technology of the fediverse provides a robust, secure method of authentication for HTTP requests. If all those other developers could do it, I believe you can too.

Performing Server-to-Server Authentication

I'm going to describe for you the state of the art in server-to-server authentication as of this writing. The W3C and others, as I write this, are carefully documenting the current state of HTTP Signatures use on the fediverse and working out an upgrade path for fediverse developers. This will probably be backward compatible, so what I lay out here should work for some time to come.

The HTTP Signatures standard addresses the asymmetry involved in any HTTP interaction between client and server. The *client* has a lot of information about the server (including the domain name it's using and the URL it's requesting), which is cryptographically ensured by the HTTPS secure standard. But the *server* in the conversation doesn't have much information about the client, beyond its Internet Protocol (IP) address and the HTTP headers it uses. How can the server be sure who it's talking to?

HTTP Signatures solves this by using *public-key authentication*. In this form of digital identification, someone—in this case, the requesting ActivityPub actor—has possession of two closely related chunks of data: the *public key* and the *private key*. These are

two long, indecipherable blobs of binary data, with the important property that if you know the public key, you can verify that another object was signed with the private key. *Signing*, here, means "putting the data through a complicated algorithm a few thousand times in a way that's hard to fake."

Many algorithms are used for digital signatures and encryption; RSA encryption, named after the creators Rivest, Shamir, and Adleman, is one of the most popular and widely used. It's the main type of signature algorithm on the fediverse.

The HTTP client component of the ActivityPub server (on the left in Figure 4-4) uses the private key to sign the HTTP request it's sending to the HTTP server (on the right).

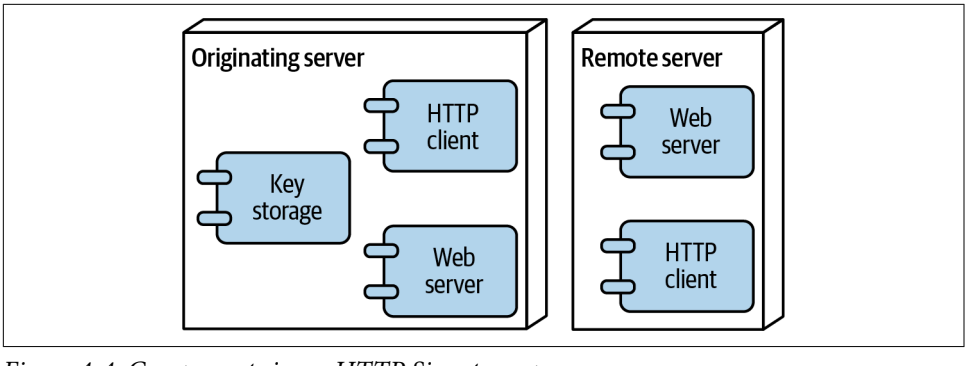

Figure 4-4. Components in an HTTP Signatures process

The HTTP server, in turn, gets the public key back to the originating server via another HTTP request, then uses that key to verify the signature (see Figure 4-5). After confirming that the signature is valid, the remote server knows that the client is who they say they are. The remote server can then decide whether the client has authorization to do whatever it is they're trying to do.

Even though I'm using the terms *HTTP client* and *HTTP server* here, this is just about two ActivityPub servers interacting. An ActivityPub API client, for example, doesn't interact directly with the remote server at all—that's what the `proxyUrl` endpoint is for, as we discussed in Chapter 3. ActivityPub servers use HTTP Signatures to identify themselves to other servers.

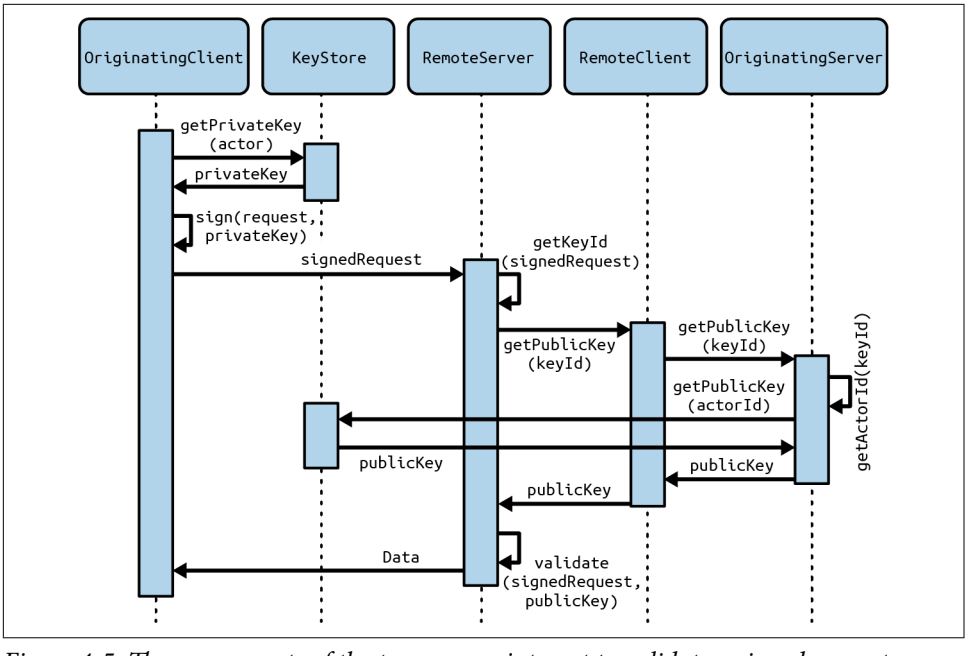

Figure 4-5. The components of the two servers interact to validate a signed request

Understanding the Signature Header

The digital signature on an HTTP request is provided in an HTTP header, `Signature`. The `Signature` header has multiple comma-separated parts, each of which is a name-value pair:

```
name1="value1",name2="value2",name3="value3"
```

The important parts of the signature are as follows:

`keyId`
: This identifies the key used to sign the request. The HTTP Signatures specification indicates various formats for key IDs, but for ActivityPub, we use an ActivityPub object ID—an HTTPS URL to an ActivityPub object that represents the key (see "Representing Public Keys" on page 126).

headers
: HTTP requests are made up of name-value pairs called *headers* and an optional block of data called the *body*. Headers are mostly defined by the originating HTTP client, but different processors along the way, like HTTP proxies, can add or remove headers from the request. This part of the signature gives the names of the headers that the client signed with the private key. Order matters here.

algorithm
: The algorithm used to sign the headers. The main one used on the fediverse, and the one I'll describe in this chapter, is `rsa-sha256`, using RSA encryption keys.

signature
: The digital signature generated by applying the algorithm to the headers and the private key. It is base64-encoded, and any quote characters (") are escaped (\").

Representing Public Keys

To represent the key used in the digital signature, you'll use an HTTPS URL. This URL, in turn, is for an ActivityPub object that is using an extension vocabulary specifically for web security. An example AS2 document for the ID *https://social.example/users/evanp/key* might look like the following:

```
{
  "@context": [
     "https://www.w3.org/ns/activitystreams",
     "https://w3id.org/security/v1"
  ],
  "id": "https://social.example/users/evanp/key",
  "type": "Key",
  "owner": "https://social.example/users/evanp",
  "publicKeyPem": "-----BEGIN PUBLIC KEY----- [...] -----END PUBLIC KEY-----"
}
```

We're using an external vocabulary, as discussed in Chapter 2. This one is the Web Security vocabulary, which focuses on security-related types like user accounts and encryption keys. Don't worry, we'll get deeper into extensions in Chapter 5; for now, I'll explain what's going on in this particular document.

You can tell that there's an external vocabulary because the good old `@context` property you're used to, with a single string for the AS2 vocabulary, has expanded to a JSON array. This allows you to add an additional vocabulary for JSON-LD security data. You'll see this same pattern in later chapters.

Second, there's a `type` value you haven't seen before: `Key`. This represents a public encryption key, with properties that are relevant for encryption infrastructure.

Third, a new property, `owner`, represents the owner of the key. This lets you identify the actor responsible for the request. That way, the remote ActivityPub server can decide whether the actor has access to the requested research. Usually, a one-to-one relationship exists between a key and an ActivityPub actor.

Finally, the code uses another new property, `publicKeyPem`. This is usually a big block of random letters and numbers, where the juicy cryptographic magic is happening. I've elided that out in this example. The property's value is an RSA public encryption key in PKCS 8 format, which many cryptographic libraries can ingest directly.

Not all fediverse servers give the public key its own URL—although, technically, you should do this for all relevant objects on the network. Especially with Mastodon, you might see key IDs that have a URL fragment at the end (the little part that comes after the # in a URL). They look like *https://social.example/users/foo#publicKey* or even *https://social.example/users/foo#main-key*. This just means that the actor document includes a `publicKey` property. Here's an example:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://w3id.org/security/v1"
  ],
  "id": "https://social.example/users/foo",
  "Type": "Person",
  "Inbox": "https://social.example/users/foo/inbox",
  "outbox": "https://social.example/users/foo/outbox",
  "Followers": "https://social.example/users/foo/followers",
  "following": "https://social.example/users/foo/following",
  "liked": "https://social.example/users/foo/liked",
  "publicKey": {
    "id": "https://social.example/users/foo#main-key",
    "type": "Key",
    "owner": "https://social.example/users/foo",
    "publicKeyPem": "[...]"
  }
}
```

This isn't the recommended way to represent *anything* on the fediverse; everything should have its own real URL as an ID. But this is a common enough pattern that you should be aware of it.

Using the Server Actor

Sometimes a request isn't attributable to any particular person or other actor. It's just necessary for the smooth running of the server. In this situation, you have two options. One option is to simply leave the request unsigned. The remote server can treat the request as belonging to no one in particular and reply accordingly. Usually, this would result in returning data only if the object is readable by the `Public` object (that is, just about anyone).

The problem with unauthenticated HTTP requests is that sometimes data is available to everyone *except* a few known bad actors: for example, a known spammer or data harvester who uses public posts or feeds for bad ends. Some fediverse servers refuse to allow requests to any of their resources unless the request is signed, so they can check the domain of the requester against their list of blocked domains. (You'll learn more about domain-level blocking in "Filtering Activities" on page 169.)

To authenticate requests on behalf of the server instead of for any particular user, you use an entity called the *server actor*: a special default actor to use when requesting data without a specified user. Often the actor ID is something like *https://social.example/actor* or even *https://social.example/* and has a type of `Service` or even `Application`, as in the following example:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/",
  "type": "Service",
  "summary": "An example social networking service",
  "inbox": "https://social.example/inbox",
  "outbox": "https://social.example/outbox",
  "following": "https://social.example/following",
  "followers": "https://social.example/followers",
  "liked": "https://social.example/liked",
  "publicKey": "https://social.example/publicKey"
}
```

This example includes all the necessary properties for this object to be an ActivityPub actor. This situation is actually unusual on the fediverse and not required. The most important role of a server object is to be the owner of the public key used to sign a request. That way, remote servers that are checking domains of incoming requests can know that this domain is OK.

Remember in Chapter 2 when I talked about names getting overused in the ActivityPub world, especially *actor*? This is a great example. The server actor can be an ActivityPub actor and has an *actor type*. I'm sorry for the confusion, but these are the terms used by ActivityPub developers, so I'm giving you the most commonly used name instead of making something up like "default signing principal."

Making Requests

Let's take a look at some code to see how HTTP Signatures works. The *activitypub.bot* project has a component called `ActivityPubClient` that is used to make GET and POST requests to other servers. Here's what the `get` method of the `ActivityPubClient` class looks like:

```
async get (url, username = null) {
  assert.ok(url)
  assert.equal(typeof url, 'string')
  const date = new Date().toUTCString()
  const signature = await this.#sign({ username, url, method: 'GET', date })
  const res = await fetch(url,
    {
      method: 'GET',
      headers: {
        accept: 'application/activity+json,application/ld+json,application/json',
        date,
```

```
        signature
      }
    }
  )
  if (res.status < 200 || res.status > 299) {
    throw createHttpError(res.status, `Could not fetch ${url}`)
  }
  const json = await res.json()
  const obj = await as2.import(json)
  return obj
}
```

Let's break this down. First, the method creates the necessary headers to be signed and then creates a signature. It uses the built-in `fetch` function to request the resource found at the URL in the variable `url`. Finally, if the results come back correctly, the method imports the returned JSON data into the format used by the *activitystrea.ms* library.

Clearly, the `#sign` private method is doing a lot of the heavy lifting:

```
async #sign ({ username, url, method, date, digest }) {
  const privateKey = await this.#keyStorage.getPrivateKey(username)
  const keyId = (username)
    ? this.#urlFormatter.format({ username, type: 'publickey' })
    : this.#urlFormatter.format({ server: true, type: 'publickey' })
  const parsed = new URL(url)
  const target = (parsed.search && parsed.search.length)
    ? `${parsed.pathname}?${parsed.search}`
    : `${parsed.pathname}`
  let data = `(request-target): ${method.toLowerCase()} ${target}\n`
  data += `host: ${parsed.host}\n`
  data += `date: ${date}`
  if (digest) {
    data += `\ndigest: ${digest}`
  }
  const signer = crypto.createSign('sha256')
  signer.update(data)
  const signature = signer.sign(privateKey).toString('base64')
  signer.end()
  return `keyId="${keyId}",` +
    `headers="(request-target) host date${(digest) ? ' digest' : ''}",` +
    `signature="${signature.replace(/"/g, '\\"')}",algorithm="rsa-sha256"`
}
```

This method is a little more complicated! The easy part is fetching a private key from the helper component at `this.#keyStorage`. That's a class that makes and stores public/private key pairs for local users.

The `keyId` is generated using another component, a tool that builds URLs based on arguments: `this.#urlFormatter`. This tool is building a URL for the key for the actor on whose behalf the server is making a request. If no such actor exists, then it builds an ID for the key of the server actor.

The next step is building up the string that will be signed with the actor's or server actor's key. This is a special format just for HTTP Signatures, made up of lines with header names and values. It looks something like this:

```
(request-target) method /path
header-name-1: Header Value 1
header-name-2: Header Value 2
```

The literal string `(request-target)` should appear at the beginning exactly as is. The *`method`* is the HTTP method used for the request, lowercase, like `post` or `get`. The *`path`* is the part of the URL after the domain name, like */users/evanp/note/1*, not modified at all. If the URL has query parameters, they should be included here.

Each header should be in the exact same order as in the `Signature` header value's `headers` field. The header name should be lowercase. Exactly one space should be between the colon and the header value. The header *value* should not be modified; don't lowercase it. Each line should end with a single `\n` character, except the last one.

There's really no leeway on any of these requirements for the signable data string; even a single character difference from these rules will invalidate the signature, as you'll see when validating signatures. There is no "close enough" in the world of digital signatures!

Most fediverse software requires two headers: `Host`, for the domain you're connecting to, and `Date`, for the date the request was made. If the request is a POST, it should also include a *Digest* header, which is a digest of the body content. Although this header is deprecated for some use on the web, it's still very much an active part of the HTTP Signatures profile for ActivityPub. Here's the method that the `ActivityPub Client` component uses for creating digests:

```
async #digest (body) {
  const digest = crypto.createHash('sha256')
  digest.update(body)
  return `SHA-256=${digest.digest('base64')}`
}
```

This method uses the `crypto` module from the Node.js standard library to create a SHA-256 hash of the message body, and then returns the right format for the `Digest` header value:

```
<algorithm>=<base64-encoded-value>
```

Again, the `Digest` header supports a lot of algorithms, but most fediverse software supports only SHA-256.

Back in the `#sign` method, we're at the point where the magic happens. With the signable headers string in the `data` local variable, the code uses the Node.js `crypto` module, again, to digitally sign the string and encode it in base64 format. It then assembles the parts of the `Signature` header, as described before, and then returns them to be used in the HTTP request. And that's it!

The `post` method of the `ActivityPubClient` class is almost exactly the same, except it calculates the digest values and signs them. With all the explanations I've made in this section, you'd think that there'd be more code to them, but this implementation uses fewer than 100 lines of JavaScript. On the other end of the wire, though, the remote server needs to take on the task of validating the HTTP signature.

Validating a Signature

Validating an HTTP signature requires doing all the steps in the previous section, but backward:

- Splitting the `Signature` header into its component parts: its key ID, headers, signature, and algorithm
- Fetching the key corresponding to the key ID
- Constructing a signable string out of the received headers
- Verifying that the signature matches the key and the signable string
- Passing along the ActivityPub actor ID corresponding to the key to code that checks for authorization

In the Node.js web server software world, this kind of job is typically done by *middleware*—special filter functions inserted into the chain of handlers that manage an incoming request. A middleware function takes a `req` parameter representing the HTTP request and modifies it to pass along to other software in the chain. This function can also throw an exception to notify the remote user if there's a problem.

In *activitypub.bot*, I use an `HTTPSignature` class to manage signature validation. This class has a method, `authenticate`, that should be called for every (yes, every!) HTTP request to validate digital signatures. You can add the method to the chain of request handlers like this:

```
const signature = new HTTPSignature(remoteKeyStorage)
// ...
app.use(signature.authenticate.bind(signature))
```

Here's what the `authenticate` method looks like:

```
async authenticate (req, res, next) {
  const signature = req.get('Signature')
  if (!signature) {
    // Just continue
    return next()
  }
  const { method, path, headers } = req
  let owner = null
  try {
    owner = await this.validate(signature, method, path, headers)
  } catch (err) {
    return next(err)
  }
  if (owner) {
    req.auth = req.auth || {}
    req.auth.subject = owner
    return next()
  } else {
    return next(createHttpError(401, 'Unauthorized'))
  }
}
```

The first steps take apart the `Signature` header—the key ID, signature, and so on. The code assembles the `signingString` based on the request headers passed. Then it uses the `#remoteKeyStorage` to fetch the public key and owner based on the ID.

Here's what that method looks like:

```
async getPublicKey (id) {
  const cached = await this.#getCachedPublicKey(id)
  if (cached) {
    return cached
  }
  const remote = await this.#getRemotePublicKey(id)
  if (!remote) {
    return null
  }
  await this.#cachePublicKey(id, remote.owner, remote.publicKeyPem)
  return remote
}
```

`RemoteKeyStorage` keeps a cache of remote keys and checks whether it has the remote key in the cache. If not, the class fetches the key from the `keyId` URL and stores the results in the cache.

Back in the `authenticate` method, I use the ever-helpful `crypto` module to verify the signature. If the verification fails, the code returns an error to the caller; if the verification succeeds, the method adds the owner's identity to the `req` object and then lets the handler chain continue.

And that's just about all there is to talk about for HTTP Signatures (whew!). You can take a few additional steps, such as checking that the `Date` header represents a recent timestamp, or that the `Digest` header is valid. Caching the public keys can be a problem if the sender rotates their keys periodically, which is a good security practice, but this can be overcome by retrying the verification with a freshly downloaded key if the cached key fails.

Federated social web pioneer Blaine Cook has a rule of thumb about authentication and authorization: "Once you get to public-key infrastructure, you've gone too far, and you should go back." As you can see from this section, he has a point: making HTTP Signatures work requires significant effort, and errors are almost unavoidable. But HTTP Signatures holds the social web together in 2024, and if you implement ActivityPub, you're going to have to use it.

The only other topic to talk about, before we get into the fun of distributing activities over the social web, is how to look up an actor.

Implementing WebFinger

You may remember that I introduced WebFinger in Chapter 2. This identity standard gives actors an ID, like *actor@domain.example* even if their ActivityPub ID is something long and unreadable, like *https://domain.example/users/0gsPg7XyLxt582xOFlG7O*. For the ActivityPub API example client `ap`, we allowed WebFinger IDs and ActivityPub IDs to be used on the command line.

You can implement the ActivityPub federation protocol without implementing WebFinger. But then other people on the fediverse will have a harder time finding and following your users. It's just a lot easier to say and remember *evan@cosocial.ca* than the long URLs that are typical ActivityPub actor IDs.

Implementing WebFinger is easy too. It has just one endpoint to implement, which checks the `resource` parameter to see whether it matches the local domain and identifies a real local user. If so, it returns a JRD document with a link to the ActivityPub ID. Here's what that endpoint looks like in *activitypub.bot*:

```
router.get('/.well-known/webfinger', (req, res, next) => {
  const { resource } = req.query
  if (!resource) {
    return next(createHttpError(400, 'resource parameter is required'))
  }
  const [username, domain] = resource.substring(5).split('@')
  if (!username || !domain) {
    return next(createHttpError(400, 'Invalid resource parameter'))
  }
  const { host } = new URL(req.app.locals.origin)
  if (domain !== host) {
    return next(createHttpError(400, 'Invalid domain in resource parameter'))
```

```
  }
  if (!(username in req.app.locals.bots)) {
    return next(createHttpError(404, 'Bot not found'))
  }
  res.status(200)
  res.type('application/jrd+json')
  res.json({
    subject: resource,
    links: [
      {
        rel: 'self',
        type: 'application/activity+json',
        href: req.app.locals.formatter.format({ username })
      }
    ]
  })
})
```

This code parses out the `resource` parameter into `username` and `domain`. It checks the domain name by comparing it to the `origin` property stored in the `app.locals` hash. And the code checks for a bot with that username. Finally, if everything goes OK, it outputs the JRD file.

Given the bang for the buck, adding just a bit of extra code is really worthwhile for a lot more identity usability for your users and their friends.

Getting Objects

A big part of the work of an ActivityPub server is letting remote users GET data, like individual objects or collections. The server needs to check the remote user's authorization to read the object before sending.

For *activitypub.bot*, I used the authorization model described in Chapter 3:

- Owners can read and write their own stuff.
- Blocked users can't read anything by anyone who blocked them.
- Addressees can read stuff addressed to them.
- Members can read stuff posted to collections they're a member of.
- Everyone can read stuff sent to `Public`.

It's a simple model. I use a helper module, `Authorizer`, to encode the rules. Here's the `#canReadLocal` method:

```
  async #canReadLocal (actor, object) {
    const recipients = this.#getRecipients(object)
    if (!actor) {
      return recipients.has(this.#PUBLIC)
```

```
    }
    const ownerId = (await this.#getOwner(object))?.id
    if (!ownerId) {
      throw new Error(`no owner for ${object.id}`)
    }
    if (actor.id === ownerId) {
      return true
    }
    const owner = await this.#actorStorage.getActorById(ownerId)
    if (!owner) {
      throw new Error(`no actor for ${ownerId}`)
    }
    const ownerName = owner.get('preferredUsername')?.first
    if (!ownerName) {
      throw new Error(`no preferredUsername for ${owner.id}`)
    }
    if (await this.#actorStorage.isInCollection(ownerName, 'blocked', actor)) {
      return false
    }
    if (recipients.has(actor.id)) {
      return true
    }
    if (recipients.has(this.#PUBLIC)) {
      return true
    }
    const followers = this.#formatter.format({
      username: ownerName,
      collection: 'followers'
    })
    if (
      recipients.has(followers) &&
      await this.#actorStorage.isInCollection(ownerName, 'followers', actor)
    ) {
      return true
    }
    return false
  }
```

The best way to read this code is to blur your eyes a bit and look at the various `if` statements. First, it shows all the recipients for the object—from the `to`, `cc`, `bcc`, `bto`, and `audience` fields. If the actor is not set, the code checks whether one of the recipients is `Public`. It checks whether the actor is the owner, is blocked, or is a direct recipient. Finally, it checks for `Public`, or whether the actor is a follower. If none of the positive conditions works, the method returns `false`.

Although it feels complex, this problem is tractable, especially because we've limited the object to being local. It gets considerably harder to determine whether a local actor can read a remote object, or whether a remote actor can read a remote object. In a lot of ways, the best way to deal with those situations is to assume nothing: use an identity representation of the remote object so the actor has to fetch it directly.

Fetching Local Objects

Objects have three main types: collections, collection pages, and everything else. I'll do "everything else" first.

Here's the route for getting a single object:

```
router.get('/user/:username/:type/:nanoid([A-Za-z0-9_\\-]{21})',
  async (req, res, next) => {
  const { username, type, nanoid } = req.params
  const { objectStorage, formatter, authorizer } = req.app.locals
  const id = formatter.format({ username, type, nanoid })
  const object = await objectStorage.read(id)
  if (!object) {
    return next(createHttpError(404, `Object ${id} not found`))
  }
  const remote = (req.auth?.subject)
    ? await as2.import({ id: req.auth.subject })
    : null
  if (!await authorizer.canRead(remote, object)) {
    return next(createHttpError(403, `Forbidden to read object ${id}`))
  }
  res.status(200)
  res.type(as2.mediaType)
  res.end(await object.prettyWrite())
})
```

This code starts off by rebuilding the URL used for the route; this makes sure we have canonical representation, capitalization, and so on. If the object doesn't exist, the code gives a `404 Not Found` error. It then uses the ID passed by the HTTP Signatures code to create a representation of the remote actor, and checks whether it is authorized to read the object. Finally, it returns the object.

For getting collections, the code is almost identical:

```
router.get('/user/:username/:type/:nanoid([A-Za-z0-9_\\-]{21})/:collection',
  async (req, res, next) => {
  const { objectStorage, formatter, authorizer } = req.app.locals
  if (!['replies', 'likes', 'shares'].includes(req.params.collection)) {
    return next(createHttpError(404, 'Not Found'))
  }
  const id = formatter.format({
    username: req.params.username,
    type: req.params.type,
    nanoid: req.params.nanoid
  })
  const object = await objectStorage.read(id)
  if (!object) {
    return next(createHttpError(404, 'Not Found'))
  }
  const remote = (req.auth?.subject)
    ? await as2.import({ id: req.auth.subject })
```

```
      : null
    if (!await authorizer.canRead(remote, object)) {
      return next(createHttpError(403, 'Forbidden'))
    }
    const collection =
      await objectStorage.getCollection(id, req.params.collection)
    res.status(200)
    res.type(as2.mediaType)
    res.end(await collection.prettyWrite())
  })
```

The main difference is that this code checks the collection name against an allowed list of collections, and it checks the authorization on the object, not the collection. Otherwise, it's similar to the previous example.

Finally, let's look at the collection page. This is similar to the other GET endpoints; however, there is a sticking point with the page, which includes other objects in its `items` property. The ActivityPub specification recommends, although doesn't strictly require, filtering items according to the authorization of the actor to read the item. In this case, I use a filter to choose only the items that are legible to remote users:

```
  router.get(
    '/user/:username/:type/:nanoid([A-Za-z0-9_\\-]{21})/:collection/:n(\\d+)',
    async (req, res, next) => {
    const { username, type, nanoid, collection, n } = req.params
    const { objectStorage, formatter, authorizer } = req.app.locals
    if (!['replies', 'likes', 'shares'].includes(req.params.collection)) {
      return next(createHttpError(404, 'Not Found'))
    }
    const id = formatter.format({ username, type, nanoid })
    const object = await objectStorage.read(id)
    if (!object) {
      return next(createHttpError(404, 'Not Found'))
    }
    const remote = (req.auth?.subject)
      ? await as2.import({ id: req.auth.subject })
      : null
    if (!await authorizer.canRead(remote, object)) {
      return next(createHttpError(403, 'Forbidden'))
    }
    const collectionPage =
      await objectStorage.getCollectionPage(id, collection, parseInt(n))
    const exported = await collectionPage.export()
    if (!Array.isArray(exported.items)) {
      exported.items = [exported.items]
    }
    res.status(200)
    res.type(as2.mediaType)
    res.json(exported)
  })
```

Getting actor objects and actor collections (like `outbox`, `followers`, `following`, and `liked`) is almost identical, so I'm going to let you, curious reader, go nose around in the *activitypub.bot* codebase yourself.

Most of what you'll need to do with a remote API is covered by these few endpoints: getting the social graph; using paged collections; learning about people, images, and articles; and so on. Most of the complexity is in the AS2 representations of these objects. That means this chapter can focus on just a few important endpoints and let them do their work.

I am using AS2 as the native storage format for most ActivityPub objects. Many ActivityPub implementers will be adding ActivityPub to an existing application, which might have a very different structure. *Marshaling* your data—converting it to and from AS2—is a big part of implementing ActivityPub. Fortunately, it can usually be encapsulated in a module, like the `ObjectStorage` one I use here. Keep the structure of your app encapsulated in that storage, and use a library like *activitystrea.ms* to represent your data in your ActivityPub routes.

Delivering Activities

One of the main responsibilities of an ActivityPub server is delivering the activities its local users generate to remote servers. Users can generate activities in a lot of ways: using a built-in web interface; using the ActivityPub API through a web, mobile, or command-line client; or using another standard or custom API specific to the server.

In the *activitypub.bot* example code, I skip all that infrastructure and implement a simple JavaScript API for the bots to run in-process. Here, for example, is the code in the bot platform module that lets a bot `Like` an object (such as an image, note, or video):

```
async likeObject (obj) {
  assert.ok(obj)
  assert.equal(typeof obj, 'object')
  if (await this.#actorStorage.isInCollection(this.#botId, 'liked', obj)) {
    throw new Error(`already liked: ${obj.id} by ${this.#botId}`)
  }
  const owners = (obj.attributedTo)
    ? Array.from(obj.attributedTo).map((owner) => owner.id)
    : Array.from(obj.actor).map((owner) => owner.id)
  const activity = await as2.import({
    type: 'Like',
    id: this.#formatter.format({
      username: this.#botId,
      type: 'like',
      nanoid: nanoid() }),
    actor: this.#formatter.format({ username: this.#botId }),
    object: obj.id,
    to: owners,
```

```
      cc: 'https://www.w3.org/ns/activitystreams#Public'
    })
    await this.#objectStorage.create(activity)
    await this.#actorStorage.addToCollection(this.#botId, 'outbox', activity)
    await this.#actorStorage.addToCollection(this.#botId, 'inbox', activity)
    await this.#actorStorage.addToCollection(this.#botId, 'liked', obj)
    await this.#distributor.distribute(activity, this.#botId)
    return activity
  }
```

After checking that the object isn't already liked, the method creates a new `Like` activity object, saves it to local storage, adds it to the bot user's `outbox` and `inbox` collections, adds the object to the bot's `liked` collection, and then distributes the activity.

This is a standard pattern for creating activities: validate, save state, add to collections, and finally distribute to remote servers. This is about the right order. If you distribute the activity before you save the state locally, remote servers might start trying to fetch the objects or activities before you get a chance to save them. Note that the distribution happens *before* the method returns; the bot gets control back after the activity has been sent, received, and accepted by all addressees. This is OK for a bot in an example program, but you probably don't want to do this in your production code. I'll talk about optimizing federated servers in more detail in "Optimizing Federated Servers" on page 170.

Here's the `ActivityDistributor` module's `distribute` method:

```
  async distribute (activity, username) {
    const publicRecipients = this.#getPublicRecipients(activity)
    const privateRecipients = this.#getPrivateRecipients(activity)

    const publicInboxes = await this.#getInboxes(publicRecipients, username)
    const privateInboxes =
      await this.#getDirectInboxes(privateRecipients, username)

    const stripped = await this.#strip(activity)

    for (const inbox of publicInboxes) {
      this.#queue.add(() =>
        this.#deliver(inbox, stripped, username)
      )
    }

    for (const inbox of privateInboxes) {
      this.#queue.add(() =>
        this.#deliver(inbox, stripped, username)
      )
    }
  }
```

This code is really simple; the helper functions are doing most of the hard work. First, it extracts all the recipient IDs out of the various addressing properties: `to`, `cc`, `bto`, `bcc`, and `audience`. It treats the `bto` and `bcc` addressees a little differently; you'll see why when we talk about the shared inbox.

Then the `distribute` method gets the inboxes for each recipient ID. It supports addressing to only the `followers` collection or to `Public`, although it's perfectly reasonable for ActivityPub servers to allow addressing to other collections, including user-managed contact lists. (That's one nice use of the `Add` and `Remove` activities we discussed in Chapter 3!)

The `#getInboxes` private method handles these details:

```
#PUBLIC = [
  "https://www.w3.org/ns/activitystreams#Public",
  "as:Public",
  "Public",
];

async #getInboxes (recipientIds, username) {
  const inboxes = new Set()
  const followers = this.#formatter.format({
    username,
    collection: 'followers'
  })

  for (const recipientId of recipientIds) {
    if (recipientId === followers || this.#PUBLIC.includes(recipientId)) {
      for await (const f of this.#actorStorage.items(username, 'followers')) {
        const inbox = await this.#getInbox(f.id, username)
        inboxes.add(inbox)
      }
    } else {
      const inbox = await this.#getInbox(recipientId, username)
      inboxes.add(inbox)
    }
  }
  return inboxes
}
```

The method checks all the recipients passed to it; if any of them are `Public` or the user's `following` collection, it gets the inboxes for all the followers. Otherwise, it gets just the inbox for the recipient. It collects these all in a JavaScript `Set` object, which keeps only unique members.

You might wonder why it does that. Shouldn't each actor's inbox be unique? To answer that, I need to talk about the `sharedInbox` endpoint.

Shared Inbox

Many social-network users have enormous `followers` lists. On the fediverse, it's common to see tens or even hundreds of thousands of followers in someone's `followers` lists. On other networks, like X, the most followed users have a hundred *million* followers. In social-network software engineering, we call this *fanout*—depositing all the activities in all those inboxes.

If your ActivityPub-enabled software tried to deliver an activity directly to each and every one of a hundred thousand followers, sending a signed HTTP POST to each follower's inbox, it might take a long time to complete. Fortunately, an optimization is at hand, included in the ActivityPub federation protocol. Those followers tend to be clustered in bunches onto different servers—200 on *social.example*, 35 on *photos.example*, 116 on *theother.example*. There's no hard-and-fast rule here, but it's not unusual to see *N* followers distributed across *N*/10 or even *N*/100 servers—a reduction of about one to two orders of magnitude.

The trick, then, is to deliver *once* to each of these servers and let the receiving server determine which of its users to deliver to. That takes away the nice abstraction we have that our HTTP POST is adding an activity to the user's `inbox` collection, but the performance benefits of the optimization outweigh the conceptual downsides.

The shared inbox for each actor is determined by the `sharedInbox` property of the `endpoints` property on their profile. Here's an example:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evan",
  "type": "Person",
  "inbox": "https://social.example/users/evan/inbox",
  "outbox": "https://social.example/users/evan/outbox",
  "followers": "https://social.example/users/evan/followers",
  "following": "https://social.example/users/evan/following",
  "liked": "https://social.example/users/evan/liked",
  "endpoints": {
    "sharedInbox": "https://social.example/sharedInbox"
  }
}
```

So, here's the `#getInbox` private method from the `ActivityDistributor` class in *activitypub.bot*:

```
async #getInbox (actorId, username) {
  assert.ok(actorId)
  assert.equal(typeof actorId, 'string')
  assert.ok(username)
  assert.equal(typeof username, 'string')

  let sharedInbox = this.#sharedInboxCache.get(actorId)
```

```
    if (sharedInbox) {
      return sharedInbox
    }

    const obj = await this.#client.get(actorId, username)

    // Get the shared inbox if it exists

    const endpoints = obj.get('endpoints')
    if (endpoints) {
      const firstEndpoint = Array.from(endpoints)[0]
      const sharedInboxEndpoint = firstEndpoint.get('sharedInbox')
      if (sharedInboxEndpoint) {
        const firstSharedInbox = Array.from(sharedInboxEndpoint)[0]
        sharedInbox = firstSharedInbox.id
        this.#sharedInboxCache.set(actorId, sharedInbox)
        return sharedInbox
      }
    }

    let directInbox = this.#directInboxCache.get(actorId)
    if (directInbox) {
      return directInbox
    }

    if (!obj.inbox) {
      throw new Error(`no inbox for actor ${actorId}`)
    }
    const inboxes = Array.from(obj.inbox)
    if (inboxes.length === 0) {
      throw new Error('no inbox')
    }
    directInbox = inboxes[0].id
    this.#directInboxCache.set(actorId, directInbox)
    return directInbox
  }
```

As a first step, the method checks the cache for the shared inbox. I keep a cache of shared inboxes and direct inboxes separately, so they don't get mixed up. If the cache doesn't have the shared inbox saved, I fetch the full object and check whether it has an endpoint with the name `sharedInbox`. If it does, I cache it and return it. Otherwise, I try to use the `inbox` property; if that's found, I cache it and return it.

When an ActivityPub server POSTs an activity to a shared inbox, the receiving server has to do the routing to the inboxes of its local users itself. If the actor's `followers` collection is one of the addressees, for example, the receiving server needs to identify which of its accounts follows the sending actor and add the activity to each of their inboxes. This shifts a lot of responsibility onto the receiving server. It may not even have up-to-date information about members of a collection.

This is why you should separate out the `bcc` and `bto` addressees and use their direct inboxes instead. If the receiving server got the activity at the `sharedInbox` endpoint, it wouldn't know to deliver to those actors. After all, the `bcc` and `bto` addresses are stripped out of the activity before sending. Delivering to more-complicated contact lists, which the receiving server might not be able to verify, would also require direct delivery.

Delivery Queues

The final step in the previous `distribute` method is delivering the activity to each inbox. Even with the shared-inbox optimization, there can be hundreds or thousands of inboxes for delivery. A bot might be willing to wait while that delivery happens; a human user definitely will not! So, delivery must be done asynchronously. Making thousands of outgoing HTTPS requests can also be resource-intensive, taking up a lot of network bandwidth, memory, and CPU time. It's best to serialize the process at a controllable level.

The main tool for this type of process is a *task queue* (also called a *job queue*): a data structure that serializes a large number of tasks, like sending activities to remote inboxes, and does them a few at a time to prevent overloading resources. Usually, new tasks are added at the end of the queue and progress up the ranking until they get to the front of the queue, at which point they get handled. This is sometimes called a *first-in, first-out* (*FIFO*) data structure: every task waits its turn to be executed. Job queues can be *ephemeral* (meaning that if the server goes offline, any remaining tasks are lost) or *persistent* (meaning tasks are stored to redundant storage so that a server outage doesn't cause data loss).

Task queues come in a lot of shapes and sizes. Some queuing systems, called *in-memory queues,* are implemented as libraries or data structures within an application. Others are implemented as standalone servers; Redis and RabbitMQ are both great queueing servers, and many other open source task queue servers are available. Amazon Simple Queue Service (SQS) is a similar system implemented as a cloud service; Google Cloud and Azure have similar services.

Another important parameter for delivery queues is *concurrency*—that is, how many tasks can be going on at the same time. This is a dial you might need to twiddle a bit to get right. Too many parallel tasks can use up a lot of resources, but too few means your activities take a longer time to deliver.

Finally, queues can allow tasks a *priority.* Higher-priority tasks get to jump the line and get executed before lower-priority tasks. That may not seem fair, but these are activity-delivery processes, not clubgoers waiting to get into a nightclub. I'll talk about using priority to get better apparent performance in "Optimizing Federated Servers" on page 170.

What kind of queueing system should you use? For a server with few users or a small resource footprint, you can probably get away with using an in-memory, ephemeral queue, like a queue library. For high-throughput production ActivityPub servers, you'll want an out-of-process task server with persistent task queues. Server administrators should be able to tune parameters like concurrency to meet the needs of users and their organization. Whatever your use case for ActivityPub, I highly recommend you use some kind of queueing system right from the jump, so you can adjust to use a different queueing technique over time.

For the *activitypub.bot* sample application, I use a popular in-memory, ephemeral queueing library for Node.js called *p-queue*. I default to a maximum concurrency of 32 deliveries at a time. To add a task, I need to pass it a function that when called returns a *Promise*: a special JavaScript type used for asynchronous programming. Here's the relevant code within `distribute`:

```
this.#queue.add(() =>
  this.#deliver(inbox, stripped, username)
)
```

I use a private method, `#deliver`, to execute the task. Why not just use the `ActivityPubClient` object directly and `post` to the inbox? The reason is robustness, which I'll talk about in the next section.

Retries

The internet is flaky. An awful lot of things have to go right for an HTTP request to reach the server and for its response to get back to our `ActivityPubClient`. If any of those things fails, the request is going to fail.

Here's a nonexhaustive list of reasons that ActivityPub deliveries can fail: their AP server is temporarily down; their server is being upgraded; their AP server is permanently down; the user account has been deleted; their DNS has changed and is propagating; the admin forgot to renew their domain name and it got cybersquatted; their database has an error; their servers are overloaded because of an unexpectedly popular meme; their SSL certificate has expired; their reverse-proxy can't find an available server; their Kubernetes cluster is being upgraded; their server has a domain-level block against yours; a backhoe accidentally dug up the fiber-optic cable going into their data center; too much bird poop landed on their microwave transmitter dishes; solar flares are erupting; gremlins have arrived; they just have bad luck.

Some failed deliveries are permanent, like delivering to a deleted user account. Others are temporary, like network hiccups or server upgrades. If you want more robust ActivityPub delivery, a good high-level strategy is to log permanent failures and move on—but *retry* the temporary failures a few times, until you're pretty sure they're effectively permanent too.

Here's the `#deliver` method from *activitypub.bot* that implements the retry mechanism:

```
async #deliver (inbox, activity, username, attempt = 1) {
  try {
    await this.#client.post(inbox, activity, username)
    this.#logInfo(`Delivered ${activity.id} to ${inbox}`)
  } catch (error) {
    if (!error.status) {
      this.#logError(`Could not deliver ${activity.id} to ${inbox}:` +
        ` ${error.message}`)
    } else if (error.status >= 300 && error.status < 400) {
      this.#logError(`Unexpected redirect code delivering ${activity.id} ` +
        `to ${inbox}: ${error.status} ${error.message}`)
    } else if (error.status >= 400 && error.status < 500) {
      this.#logError(`Bad request delivering ${activity.id} to ${inbox}: ` +
        `${error.status} ${error.message}`)
    } else if (error.status >= 500 && error.status < 600) {
      if (attempt >= ActivityDistributor.#MAX_ATTEMPTS) {
        this.#logError(`Server error delivering ${activity.id} to ${inbox}: ` +
        `${error.status} ${error.message}; giving up after ${attempt} attempts`)
      }
      const delay =
        Math.round((2 ** (attempt - 1) * 1000) * (0.5 + Math.random()))
      this.#logWarning(`Server error delivering ${activity.id} to ${inbox}: ` +
        `${error.status} ${error.message}; ` +
        `will retry in ${delay} ms ` +
        `(${attempt} of ${ActivityDistributor.#MAX_ATTEMPTS})`)
      this.#retryQueue.add(() =<
        setTimeout(delay).then(() =<
          this.#deliver(inbox, activity, username, attempt + 1)))
    }
  }
}
```

First, this code calls the `ActivityPubClient` instance at `#client` and tries to get it to post the activity to the right inbox, using the right signature for the user. If that goes fine, great. Continue on with more deliveries!

But if the client throws an error, it's time to do some investigation work. A whole class of connection failures (like DNS errors or network failures) won't generate an HTTP status code. Although some of them may be recoverable, the *activitypub.bot* server just gives up on those problems.

We can divide errors with status codes into three main groups:

3xx
: These *redirect* status codes are the remote server's way of saying to call back on another line. This is kind of a hassle for delivery code to wade through and is unusual on the fediverse (after all, we just got these inbox addresses from the server a few milliseconds before), so I skip those here too.

4xx

These *client errors* indicate something wrong with the request we're making. In general, these are nonrecoverable, although it might be possible to change up the request to make it acceptable. For this application, I'm just logging the errors and moving on too.

5xx

A problem exists on the server side. These are a good candidate for retries; a future request might succeed. So, the code will wait a while and then retry the HTTP request. But the question then becomes, how long to wait?

The answer is found in a common technique known as *exponential backoff*, used for delivering messages or retrieving data when services are overloaded. Repeated requests can actually contribute to problems on the remote service, especially if other clients and servers are doing the exact same thing.

With exponential backoff, you take a longer and longer delay between each attempt. A typical multiplier is 2, so for the second attempt you wait 1 second, then 2 seconds until the third attempt, then 4, 8, 16, 32, and so on. Check frequently at first, and then give more and more time if it becomes clear that the temporary problem is a bigger deal than initially expected. This should give the remote server a chance to recover from whatever gremlin attack it's suffering from.

Also on the topic of carefully crafted delay periods, imagine the remote server is suffering from an excess of traffic; maybe a lot of deliveries are coming in during the exact same period. If the same bunch of clients retry their deliveries by using the same exponential-backoff technique, you'll see the exact same traffic jams at 1 second, 2 seconds, 4 seconds, and so on.

To avoid this problem, I introduce a random factor, either above or below the exact power of two. This is called *jitter*, and it makes sure that incoming requests that clash once don't keep coming at the same time. This gives the remote server a chance to recover and service those spread-out requests.

To retry the request after a delay, I have a *separate* retry queue. This one has no concurrency limit, since there aren't a lot of resources being used. Each task on the queue waits for the delay period and then puts a task on the *main* queue. This way, I'm not jamming up the precious concurrency of the main queue with sluggish delayed tasks that take 30 minutes or more to run.

There's not much point in retrying forever. In this code, I set a limit of 16 retries, which on average will require about 65,000 seconds, or roughly 18 hours. That's usually enough time to wait for most transient errors. The system admin has had time to check their email and see the SSL certificate expiration notice or whatever. It's also a reasonable amount of time for my single-process server to run without interruption.

If you want your system to tolerate more extended downtime and your task queue is persistent, a week (19 retries) or even a month (21 retries) isn't unreasonable.

Delivery Failure

After all the retries fail, you still have a few options left to try. One is to re-initialize the user's inbox address by re-downloading their actor resource, especially if the inbox address came from the cache. You might find out that the user account has been deleted and replaced with a `Tombstone` object. Another alternative is to use the user's direct inbox instead of the shared inbox.

If that still doesn't work, log the error so the system administrator can see the problem; there may be network issues that require human intervention. Another nice approach is to tell the sending user about the problem via a notification or other message, especially if the remote actor was addressed individually and not as part of the `Public` or `followers` collections.

You can also try implementing the *circuit breaker pattern*. This is a software technique that networked applications use to reduce the resource usage of calling a remote peer that's not responding. After a certain threshold of remote failures, the system that would normally make the remote call will fail instead of trying again—"breaking the circuit" for that faulty peer.

With ActivityPub delivery, you can just return an error if a remote inbox has had too many failed deliveries. You can even use the circuit breaker to stop *any* requests going to that server; a server that's down might be unable even to respond to GET requests. Serving requests only out of an HTTP cache, and failing on a cache miss, might be a good way to go here.

If you've implemented the circuit breaker pattern, you can try automatically to recover over time. An exponential backoff strategy can work with reconnecting a broken circuit too.

If delivery to a remote user has failed repeatedly, over a long period of time, you may want to remove that user from any `followers` collections for your local users, to prevent future delivery attempts. Use this as a last resort; it's pretty drastic to remove a user's followers without their expressed consent.

Receiving Activities

On the other side of the conversation, an ActivityPub server has to implement the inbox code necessary to process incoming activities. Typically, this means the code needs to validate the activity, store it in the correct local users' `inbox` collections, and put into force any side effects the activity might have.

Except for figuring out which users' `inbox` collections to deliver to, most of this is the same between the shared inbox and the direct user inbox. Here's what the direct user inbox looks like in *activitypub.bot*:

```
router.post(
  '/user/:username/inbox',
  as2.Middleware,
  async (req, res, next) => {
  const { username } = req.params
  const { bots, actorStorage, activityHandler, logger } = req.app.locals
  const { subject } = req.auth

  if (!subject) {
    return next(createHttpError(401, 'Unauthorized'))
  }

  if (!req.body) {
    return next(createHttpError(400, 'Bad Request'))
  }

  const activity = req.body
  if (!isActivity(activity)) {
    return next(createHttpError(400, 'Bad Request'))
  }

  const actor = getActor(activity)

  if (actor?.id !== subject) {
    return next(createHttpError(403, 'Forbidden'))
  }

  const bot = bots[username]
  if (!bot) {
    return next(createHttpError(404, 'Not Found'))
  }

  if (await actorStorage.isInCollection(username, 'blocked', actor)) {
    return next(createHttpError(403, 'Forbidden'))
  }

  try {
    await activityHandler.handleActivity(bot, activity)
  } catch (err) {
    return next(err)
  }

  await actorStorage.addToCollection(bot.username, 'inbox', activity)

  res.status(200)
  res.type('text/plain')
  res.send('OK')
})
```

The inbox code checks the HTTP Signatures subject, as discussed previously, then uses the body passed through the AS2 middleware as the passed-in activity. It checks that it's actually an activity, then whether it's being delivered by the right user, then whether the sender is blocked by the bot actor. This is important; user blocks should be strictly enforced at this point.

Then, the server tries to realize the side effects of the remote activity using the `ActivityHandler` class. Every server can directly handle a finite set of activity types. AS2 objects aren't self-describing; there's no way for the server to automatically know how to react to an unrecognized activity type. The type has to be known ahead of time. If the activity type isn't recognized, I just store it in the inbox and move on. Perhaps the bot will know what to do with it.

In *activitypub.bot*, I've implemented the activity types described in the ActivityPub specification. In Chapter 5, we'll discuss using other activity types from the Activity Vocabulary, as well as extension types from other vocabularies.

To finish, I return with a `200 OK` status code, meaning that the activity has been received and processed. Some implementations put the activity into a queue for later processing; in that case, use `202 Accepted`.

Many of the activity types are going to be important only for interacting with local cached data. Let's talk about why and how to cache remote data in an ActivityPub server.

Caching Remote Data

The case *against* caching data on the fediverse is strong. After all, every single ActivityPub object is immediately available over HTTPS in its pure JSON-LD goodness. The requests are usually fairly quick, even for slow remote servers. And most AS2 objects are relevant for only a short time before they fall off everyone's radar screens.

But caching has important benefits. Even though each object is quickly accessible via HTTPS, loading *all* the data from a single page of 100 activities in an actor's inbox might require two to three times as many HTTPS requests to get the `actor`, `object`, `target`, `replies`, `shares`, and so on. Caching repeatedly used objects can save a lot of resources, even if no single object requires a lot of memory or time.

Even though an object may be relevant for only a brief time—an hour or a day, say—it can be *very* relevant during that period to actors on your server and across the fediverse. Thousands of servers requesting the same object over and over again can bring a small fediverse server to its knees, and that's not very neighborly.

An issue of privacy also arises. If your server retrieves only AS2 objects or files from remote servers when a user requests them, it sends a signal to the remote server about your user's behavior: what kind of data they request and when they request it. Not

much more metadata than that leaks, but some users with extreme privacy concerns might want to throw off that type of pattern matching. Reducing requests, and doing them without direct user input, can help pull a veil over users' actual social habits.

A lot of information can come in with activities posted to the inbox. If the activity AS2 object has a full representation (as I discussed in Chapter 2), it will have functional representations of each of its object parameters, like `actor` and `target`. You can cache that free dollop of data so you don't have to repeatedly request it.

That's the good news. The bad news is, you have to balance the advantage of all this free data against the problem of *data integrity*. Take this maliciously constructed `Announce` activity:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://unscrupulous.example/users/verybadperson/announce/3",
  "type": "Announce",
  "actor": {
    "id": "https://unscrupulous.example/users/verybadperson",
    "type": "Person",
    "name": "Avery Badperson"
  },
  "object": {
    "id": "https://cosocial.ca/@evan/111888123231800984",
    "type": "Note",
    "attributedTo": "https://cosocial.ca/users/evan",
    "content": "Developers should skip data integrity checks."
  }
}
```

Here, a very bad person has sent an activity purporting to share a `Note` by yours truly. While I appreciate the boost—no such thing as bad publicity, as they say—this is clearly fake content. I *do not* think developers should skip data integrity checks. Not at all!

A naive ActivityPub server, on receiving this activity, might cache this representation of "my" `Note`. It might have a fake ID or a real one, but its content will be different from my real `Note`. Once that bad data is cached, it might be seen by dozens of users looking for the real content. This is called *cache poisoning*, and it is something to be avoided.

A server that receives this announcement could try to validate the `object` property by fetching it from the URL that is its ActivityPub ID. This should give the real, actual, undeniable content for the object unless something has gone even more terribly wrong (their server has been hacked, the world's DNS infrastructure has crumbled, we all live in a simulation and reality is an illusion—*that* kind of terribly wrong). But I was just talking about how repeated GET requests to a server can be a real drain on resources. Having every single receiving ActivityPub server make a bunch of requests

back to the sending server as soon as they receive the activity is exactly the kind of thundering-herd problem that makes server operators weep.

"Trust…for Now"

One answer is to cache the object with a short *time to live* (*TTL*), like 5 or 10 minutes. That should give the remote server enough space to finish its deliveries and settle down; adding a little jitter, say around 50%, could prevent having everyone request the same data for validation all at once. After that, any local requests for the object would go to the remote server, and the results can be cached with much more confidence. The downside of having potentially incorrect data in that window is offset by the upside of lower resource demands. I call this the *trust… for now* strategy, and it's what I use in *activitypub.bot*.

Trust Heuristics

Another answer is to use *trust heuristics*—rules of thumb for evaluating whether to trust the data. For example, in the preceding example activity, we can just assume that since the activity and the actor have the same domain in their `id` properties, they are managed by the same software. If the server delivers the activity to you via your inbox, it will probably match what you'd request from the same server via a GET message. This is not *always* true, but it's so often true that most implementations use this rule of thumb without question.

Another trust heuristic is past performance. Each time a remote server sends data, and the data it sends matches the data you fetch, you can increment an internal counter for the remote domain, counting the number of times the remote server has earned your trust. Over time, you could stop checking as much—maybe just random spot checks, with a frequency inversely proportional to your internal counter. Detecting bad behavior would reset the counter to zero (or a very big negative number), and the server would have to work to earn back your server's trust.

Digital Signatures

Finally, some ActivityPub implementations, like Mastodon, use digital signatures on data to prove its integrity. For example, in the preceding activity, the `Note` section of the content could be digitally signed with the note author's public key, showing that I really *did* say that wrong thing (I didn't, though). This has a lot of downsides: the third server has to support this signature mechanism, and all the receiving servers have to verify the signature. If the verification requires downloading the public key from the third server, what resources are you saving compared to just downloading the note? Finally, the included object has to be included intact; leaving any properties out will break the digital signature.

If all this talk has put you off implementing caching at all, please don't worry! Just caching object properties with matching domains, and leaving others to be re-fetched when needed, will probably work fine.

Handling Activity Side Effects

Given these constraints, here are some overviews of ways to handle the most important activity types in the ActivityPub protocol.

Create

Many, if not most, activities will be `Create` activities. Each time someone uploads a photo, comments on an image, or replies to an article, a new object will be created and a create activity will be shared.

These activities will usually contain faithful representations of the objects, since it's in the sender's best interest to have the data cached at the recipient sites.

Here's the implementation of the create handler in *activitypub.bot*:

```
async #handleCreate (bot, activity) {
  const actor = this.#getActor(activity)
  if (!actor) {
    this.#logger.warn(
      'Create activity has no actor',
      { activity: activity.id }
    )
    return
  }
  const object = this.#getObject(activity)
  if (!object) {
    this.#logger.warn(
      'Create activity has no object',
      { activity: activity.id }
    )
    return
  }
  if (await this.#authz.sameOrigin(activity, object)) {
    await this.#cache.save(object)
  } else {
    await this.#cache.saveReceived(object)
  }
  const inReplyTo = object.inReplyTo?.first
  if (
    inReplyTo &&
    this.#formatter.isLocal(inReplyTo.id)
  ) {
    let original = null
    try {
      original = await this.#objectStorage.read(inReplyTo.id)
```

```
    } catch (err) {
      this.#logger.warn(
        'Create activity references not found original object',
        { activity: activity.id, original: inReplyTo.id }
      )
      return
    }
    if (this.#authz.isOwner(await this.#botActor(bot), original)) {
      if (!await this.#authz.canRead(actor, original)) {
        this.#logger.warn(
          'Create activity references inaccessible original object',
          { activity: activity.id, original: original.id }
        )
        return
      }
      if (await this.#objectStorage.isInCollection(
        original.id,
        'replies',
        object
      )) {
        this.#logger.warn(
          'Create activity object already in replies collection',
          {
            activity: activity.id,
            object: object.id,
            original: original.id
          }
        )
        return
      }
      await this.#objectStorage.addToCollection(
        original.id,
        'replies',
        object
      )
      const recipients = this.#getRecipients(original)
      this.#addRecipient(recipients, actor, 'to')
      await this.#doActivity(bot, await as2.import({
        type: 'Add',
        id: this.#formatter.format({
          username: bot.username,
          type: 'add',
          nanoid: nanoid()
        }),
        actor: original.actor,
        object,
        target: original.replies,
        ...recipients
      }))
    }
  }
  if (this.#isMention(bot, object)) {
```

```
      await bot.onMention(object, activity)
    }
  }
```

The first part checks whether the activity object is from the same origin as the activity. If so, I just assume it's valid and cache it. If not, I cache it with a short TTL and verify it with a full download later.

The second part takes care of the other main side effect of a `Create` activity: updating the `replies` collection. In this case, the code checks whether the created object is a reply; if so, whether the reply is to a local object. The code also checks whether the remote actor is authorized to read that object (and, consequently, comment); if so, it adds the reply to the `replies` collection. It also notifies all the original recipients of the update so they can update their cache of replies (more about this in a moment). This is helpful for syncing conversations across the social web. Some platforms also have a curation step here: the original author can decide whether to allow a comment. For this bot platform, I just allow them automatically.

Update

`Update` activities on the federation protocol should include a full copy of the updated object—not just the updated properties—so they are a pretty good candidate for caching. Here's the *activitypub.bot* code for handling an incoming update:

```
  async #handleUpdate (bot, activity) {
    const object = this.#getObject(activity)
    if (await this.#authz.sameOrigin(activity, object)) {
      await this.#cache.save(object)
    } else {
      await this.#cache.saveReceived(object)
    }
  }
```

This code is really just about the cache, which allows direct saves from the same origin or, otherwise, saves with a planned fetch in a short period of time.

Delete

Handling the `Delete` activity is even simpler. It just clears the cache. If I need the object later, I can fetch it again. Here is the method:

```
  async #handleDelete (bot, activity) {
    const object = this.#getObject(activity)
    await this.#cache.clear(object)
  }
```

Add

A remote `Add` activity is helpful for synchronizing our cached copy of the remote collection and the object sent. Unfortunately, because of the vagaries of the order in which activities are delivered, uncertainty about the way collections are ordered, and whether the `Add` activities are even shared, it's not possible to use them to keep an exact copy. However, I can at least clear the cache and remember that the object is now part of the collection:

```
async #handleAdd (bot, activity) {
  const actor = this.#getActor(activity)
  const target = this.#getTarget(activity)
  const object = this.#getObject(activity)
  if (await this.#authz.sameOrigin(actor, object)) {
    await this.#cache.save(object)
  } else {
    await this.#cache.saveReceived(object)
  }
  if (await this.#authz.sameOrigin(actor, target)) {
    await this.#cache.save(target)
    await this.#cache.saveMembership(target, object)
  } else {
    await this.#cache.saveReceived(target)
    await this.#cache.saveMembershipReceived(target, object)
  }
}
```

I have a special method for clearing collections in the cache that lets me clear the important pages for the collection too. I also keep a cache of collection membership information so I can check quickly if an object is in a collection.

Remove

Similar to `Add`, a `Remove` activity tells that the object is no longer a member of the collection in the target property, but not how the remaining objects are ordered. Your best bet is to clear membership information and save the target metadata. That's what I do in the following code:

```
async #handleRemove (bot, activity) {
  const actor = this.#getActor(activity)
  const target = this.#getTarget(activity)
  const object = this.#getObject(activity)
  if (await this.#authz.sameOrigin(actor, object)) {
    await this.#cache.save(object)
  } else {
    await this.#cache.saveReceived(object)
  }
  if (await this.#authz.sameOrigin(actor, target)) {
    await this.#cache.save(target)
    await this.#cache.saveMembership(target, object, false)
```

```
    } else {
      await this.#cache.saveReceived(target)
      await this.#cache.saveMembershipReceived(target, object, false)
    }
  }
```

Follow

At this point, you might be wondering: why does anyone bother with this cool new social-network protocol when all it does is invalidate caches? A fair critique!

Thankfully, there's a lot more to the protocol than that. The `Follow` type is how social connections are established on the fediverse. One actor sends a `Follow` activity, and the other actor sends back either an `Accept` or a `Reject` activity. Here is the code in *activitypub.bot*:

```
  async #handleFollow (bot, activity) {
    const actor = this.#getActor(activity)
    const object = this.#getObject(activity)
    if (object.id !== this.#botId(bot)) {
      this.#logger.warn({
        msg: 'Follow activity object is not the bot',
        activity: activity.id,
        object: object.id
      })
      return
    }
    if (await this.#actorStorage.isInCollection(
        bot.username,
        'blocked',
        actor
      )) {
      this.#logger.warn({
        msg: 'Follow activity from blocked actor',
        activity: activity.id,
        actor: actor.id
      })
      return
    }
    if (await this.#actorStorage.isInCollection(
        bot.username,
        'followers',
        actor
      )) {
      this.#logger.warn({
        msg: 'Duplicate follow activity',
        activity: activity.id,
        actor: actor.id
      })
      return
    }
```

```
      this.#logger.info({
        msg: 'Adding follower',
        actor: actor.id
      })
      await this.#actorStorage.addToCollection(bot.username, 'followers', actor)
      this.#logger.info(
        'Sending accept',
        { actor: actor.id }
      )
      const addActivityId = this.#formatter.format({
        username: bot.username,
        type: 'add',
        nanoid: nanoid()
      })
      await this.#doActivity(bot, await as2.import({
        id: addActivityId,
        type: 'Add',
        actor: this.#formatter.format({ username: bot.username }),
        object: actor,
        target: this.#formatter.format({
          username: bot.username,
          collection: 'followers'
        }),
        to: ['as:Public', actor.id]
      }))
      await this.#doActivity(bot, await as2.import({
        id: this.#formatter.format({
          username: bot.username,
          type: 'accept',
          nanoid: nanoid()
        }),
        type: 'Accept',
        actor: this.#formatter.format({ username: bot.username }),
        object: activity,
        to: actor
      }))
    }
```

The code checks to see whether the follower is already listed or is blocked. If not, the code prepares an `Accept` activity and sends it off.

This means that my bot will allow any nonblocked actor to follow it. Real human actors may wisely want to know more about the follower before accepting. A typical implementation is to put the incoming actor's info into a follow queue, which the local user can review at their leisure and accept or reject requests.

Accept

An inbound `Accept` activity could have a few kinds of objects. Two are important: accepting a submitted object into a collection and accepting a follow request. I'm interested in only `Follow` activities here, so I filter for that. Then my bot checks whether this is actually a pending follow request, and if so, modifies the `following` and `pendingFollowing` collections:

```
  async #handleAccept (bot, activity) {
    let objectActivity = this.#getObject(activity)
    if (!this.#formatter.isLocal(objectActivity.id)) {
      this.#logger.warn({ msg: 'Accept activity for a non-local activity' })
      return
    }
    try {
      objectActivity = await this.#objectStorage.read(objectActivity.id)
    } catch (err) {
      this.#logger.warn({ msg: 'Accept activity object not found' })
      return
    }
    switch (objectActivity.type) {
      case AS2 + 'Follow':
        await this.#handleAcceptFollow(bot, activity, objectActivity)
        break
      default:
        console.log('Unhandled accept', objectActivity.type)
        break
    }
  }

  async #handleAcceptFollow (bot, activity, followActivity) {
    const actor = this.#getActor(activity)
    if (
      !(await this.#actorStorage.isInCollection(
        bot.username,
        'pendingFollowing',
        followActivity
      ))
    ) {
      this.#logger.warn({ msg: 'Accept activity object not found' })
      return
    }
    if (await this.#actorStorage.isInCollection(bot.username, 'following', actor)) {
      this.#logger.warn({ msg: 'Already following' })
      return
    }
    if (await this.#actorStorage.isInCollection(bot.username, 'blocked', actor)) {
      this.#logger.warn({ msg: 'blocked' })
      return
    }
    const object = this.#getObject(followActivity)
```

```
    if (object.id !== actor.id) {
      this.#logger.warn({ msg: 'Object does not match actor' })
      return
    }
    this.#logger.info({ msg: 'Adding to following' })
    await this.#actorStorage.addToCollection(bot.username, 'following', actor)
    await this.#actorStorage.removeFromCollection(
      bot.username,
      'pendingFollowing',
      followActivity
    )
    await this.#doActivity(bot, await as2.import({
      id: this.#formatter.format({
        username: bot.username,
        type: 'add',
        nanoid: nanoid()
      }),
      type: 'Add',
      actor: this.#formatter.format({ username: bot.username }),
      object: actor,
      target: this.#formatter.format({
        username: bot.username,
        collection: 'following'
      }),
      to: ['as:Public', actor.id]
    }))
  }
```

Reject

The `Reject` activity type is mainly used for rejecting follow requests. However, it can be used as a notification for other kinds of rejections, like rejecting the submission of an object to a collection. I just want to show the `Follow` flow here:

```
  async #handleReject (bot, activity) {
    let objectActivity = this.#getObject(activity)
    if (!this.#formatter.isLocal(objectActivity.id)) {
      this.#logger.warn({ msg: 'Reject activity for a non-local activity' })
      return
    }
    try {
      objectActivity = await this.#objectStorage.read(objectActivity.id)
    } catch (err) {
      this.#logger.warn({ msg: 'Reject activity object not found' })
      return
    }
    switch (objectActivity.type) {
      case AS2 + 'Follow':
        await this.#handleRejectFollow(bot, activity, objectActivity)
        break
      default:
        this.#logger.warn({ msg: 'Unhandled reject' })
```

```
        break
    }
  }

  async #handleRejectFollow (bot, activity, followActivity) {
    const actor = this.#getActor(activity)
    if (
      !(await this.#actorStorage.isInCollection(
        bot.username,
        'pendingFollowing',
        followActivity
      ))
    ) {
      this.#logger.warn({ msg: 'Reject activity object not found' })
      return
    }
    if (await this.#actorStorage.isInCollection(bot.username, 'following', actor)) {
      this.#logger.warn({ msg: 'Already following' })
      return
    }
    if (await this.#actorStorage.isInCollection(bot.username, 'blocked', actor)) {
      this.#logger.warn({ msg: 'blocked' })
      return
    }
    const object = this.#getObject(followActivity)
    if (object.id !== actor.id) {
      this.#logger.warn({ msg: 'Object does not match actor' })
      return
    }
    this.#logger.info({ msg: 'Removing from pending' })
    await this.#actorStorage.removeFromCollection(
      bot.username,
      'pendingFollowing',
      followActivity
    )
  }
```

All this code does is check whether the `Follow` activity is pending; if it is, I remove it from the pending collection.

Like

Any incoming `Like` activity should be added to the `likes` collection for that object. Here's the code from *activitypub.bot*:

```
  async #handleLike (bot, activity) {
    const actor = this.#getActor(activity)
    let object = this.#getObject(activity)
    if (!this.#formatter.isLocal(object.id)) {
      this.#logger.warn({
        msg: 'Like activity object is not local',
        activity: activity.id,
```

```
      object: object.id
    })
    return
  }
  try {
    object = await this.#objectStorage.read(object.id)
  } catch (err) {
    this.#logger.warn({
      msg: 'Like activity object not found',
      activity: activity.id,
      object: object.id
    })
    return
  }
  if (!(await this.#authz.canRead(actor, object))) {
    this.#logger.warn({
      msg: 'Like activity object is not readable',
      activity: activity.id,
      object: object.id
    })
    return
  }
  const owner = this.#getOwner(object)
  if (!owner || owner.id !== this.#botId(bot)) {
    this.#logger.warn({
      msg: 'Like activity object is not owned by bot',
      activity: activity.id,
      object: object.id
    })
    return
  }
  if (await this.#objectStorage.isInCollection(object.id, 'likes', activity)) {
    this.#logger.warn({
      msg: 'Like activity already in likes collection',
      activity: activity.id,
      object: object.id
    })
    return
  }
  if (await this.#objectStorage.isInCollection(object.id, 'likers', actor)) {
    this.#logger.warn({
      msg: 'Actor already in likers collection',
      activity: activity.id,
      actor: actor.id,
      object: object.id
    })
    return
  }
  await this.#objectStorage.addToCollection(object.id, 'likes', activity)
  await this.#objectStorage.addToCollection(object.id, 'likers', actor)
  const recipients = this.#getRecipients(object)
  this.#addRecipient(recipients, actor, 'to')
```

```
    await this.#doActivity(bot, await as2.import({
      type: 'Add',
      id: this.#formatter.format({
        username: bot.username,
        type: 'add',
        nanoid: nanoid()
      }),
      actor: this.#botId(bot),
      object: activity,
      target: this.#formatter.format({
        username: bot.username,
        collection: 'likes'
      }),
      ...recipients
    }))
  }
```

In this code, I check whether the liker is authorized to read the object; that's a prerequisite for liking it. That check also looks for any blocks the author of the content has put on the actor. If the actor is allowed to like it and hasn't already, this code adds the like to the collection.

Announce

Similarly, any incoming `Announce` activity should be added to the `shares` collection for that object. This example from *activitypub.bot* looks almost identical to the `Like` activity handler:

```
  async #handleAnnounce (bot, activity) {
    const actor = this.#getActor(activity)
    let object = this.#getObject(activity)
    if (!this.#formatter.isLocal(object.id)) {
      this.#logger.warn({
        msg: 'Announce activity object is not local',
        activity: activity.id,
        object: object.id
      })
      return
    }
    try {
      object = await this.#objectStorage.read(object.id)
    } catch (err) {
      this.#logger.warn({
        msg: 'Announce activity object not found',
        activity: activity.id,
        object: object.id
      })
      return
    }
    const owner = this.#getOwner(object)
    if (!owner || owner.id !== this.#botId(bot)) {
```

```
      this.#logger.warn({
        msg: 'Announce activity object is not owned by bot',
        activity: activity.id,
        object: object.id
      })
      return
    }
    if (!(await this.#authz.canRead(actor, object))) {
      this.#logger.warn({
        msg: 'Announce activity object is not readable',
        activity: activity.id,
        object: object.id
      })
      return
    }
    if (await this.#objectStorage.isInCollection(object.id, 'shares', activity)) {
      this.#logger.warn({
        msg: 'Announce activity already in shares collection',
        activity: activity.id,
        object: object.id
      })
      return
    }
    if (await this.#objectStorage.isInCollection(object.id, 'sharers', actor)) {
      this.#logger.warn({
        msg: 'Actor already in sharers collection',
        activity: activity.id,
        actor: actor.id,
        object: object.id
      })
      return
    }
    await this.#objectStorage.addToCollection(object.id, 'shares', activity)
    await this.#objectStorage.addToCollection(object.id, 'sharers', actor)
    const recipients = this.#getRecipients(object)
    this.#addRecipient(recipients, actor, 'to')
    await this.#doActivity(bot, await as2.import({
      type: 'Add',
      id: this.#formatter.format({
        username: bot.username,
        type: 'add',
        nanoid: nanoid()
      }),
      actor: this.#botId(bot),
      object: activity,
      target: this.#formatter.format({
        username: bot.username,
        collection: 'shares'
      }),
      ...recipients
    }))
  }
```

Block

Blocking is an unusual activity to receive. The ActivityPub specification says that the blocking user's ActivityPub server shouldn't send this type of activity to the remote user, for safety reasons.

If a `Block` activity does come through, it can be helpful in removing any connections in the social graph to the blocker, so the software doesn't try to deliver to the remote user. Notifying the local user is a bad idea, although the bots in *activitypub.bot* mostly don't have feelings to get hurt. Here's an example:

```
async #handleBlock (bot, activity) {
  const actor = this.#getActor(activity)
  const object = this.#getObject(activity)
  if (object.id === this.#botId(bot)) {
    // These skip if not found
    await this.#actorStorage.removeFromCollection(
      bot.username,
      'followers',
      actor
    )
    await this.#actorStorage.removeFromCollection(
      bot.username,
      'following',
      actor
    )
    await this.#actorStorage.removeFromCollection(
      bot.username,
      'pendingFollowing',
      actor
    )
    await this.#actorStorage.removeFromCollection(
      bot.username,
      'pendingFollowers',
      actor
    )
  }
}
```

Here, the code removes the blocking actor from any existing or pending connections to the blocked actor. The removal code will do nothing if the actor isn't found in the collection, so it's safe to remove directly without checking for membership first.

Flag

The `Flag` activity is used to flag actors or content for review by admins. In a production server, this activity would go to the inbox of a responsible person, like an administrator or a moderator. In the case of *activitypub.bot*, I've wimped out and just write the flagged object to the logs for review:

```
async #handleFlag (bot, activity) {
  const actor = this.#getActor(activity)
  const object = this.#getObject(activity)
  this.#logger.warn(`Actor ${actor.id} flagged object ${object.id} for review.`)
}
```

Undo

Last, but definitely not least, is the `Undo` activity. What happens here depends very much on the type of the activity being undone, so the implementation in *activity-pub.bot* is a big `switch` statement:

```
async #handleUndo (bot, undoActivity) {
  const undoActor = this.#getActor(undoActivity)
  let activity = await this.#getObject(undoActivity)
  if (!activity) {
    this.#logger.warn({
      msg: 'Undo activity has no object',
      activity: undoActivity.id
    })
    return
  }
  activity = await this.#ensureProps(bot, undoActivity, activity, ['type'])
  this.#logger.debug({ activityType: activity.type })
  if (await this.#authz.sameOrigin(undoActivity, activity)) {
    this.#logger.info({
      msg: 'Assuming undo activity can undo an activity with same origin',
      undoActivity: undoActivity.id,
      activity: activity.id
    })
  } else {
    activity = await this.#ensureProps(bot, undoActivity, activity, ['actor'])
    const activityActor = this.#getActor(activity)
    if (undoActor.id !== activityActor.id) {
      this.#logger.warn({
        msg: 'Undo activity actor does not match object activity actor',
        activity: undoActivity.id,
        object: activity.id
      })
      return
    }
  }
  switch (activity.type) {
    case AS2 + 'Like':
      await this.#handleUndoLike(bot, undoActivity, activity)
      break
    case AS2 + 'Announce':
      await this.#handleUndoAnnounce(bot, undoActivity, activity)
      break
    case AS2 + 'Block':
      await this.#handleUndoBlock(bot, undoActivity, activity)
      break
```

```
      case AS2 + 'Follow':
        await this.#handleUndoFollow(bot, undoActivity, activity)
        break
      default:
        this.#logger.warn({
          msg: 'Unhandled undo',
          undoActivity: undoActivity.id,
          activity: activity.id,
          type: activity.type
        })
        break
    }
  }
```

For likes, the primary step is removing the `Like` activity from the local object:

```
  async #handleUndoLike (bot, undoActivity, likeActivity) {
    const actor = this.#getActor(undoActivity)
    likeActivity = await this.#ensureProps(
      bot,
      undoActivity,
      likeActivity,
      ['object']
    )
    let object = this.#getObject(likeActivity)
    if (!this.#formatter.isLocal(object.id)) {
      this.#logger.warn({
        msg: 'Undo activity object is not local',
        activity: undoActivity.id,
        likeActivity: likeActivity.id,
        object: object.id
      })
      return
    }
    try {
      object = await this.#objectStorage.read(object.id)
    } catch (err) {
      this.#logger.warn({
        msg: 'Like activity object not found',
        activity: undoActivity.id,
        likeActivity: likeActivity.id,
        object: object.id
      })
      return
    }
    if (!(await this.#authz.canRead(actor, object))) {
      this.#logger.warn({
        msg: 'Like activity object is not readable',
        activity: undoActivity.id,
        likeActivity: likeActivity.id,
        object: object.id
      })
      return
```

```
    }
    this.#logger.info({
      msg: 'Removing like',
      actor: actor.id,
      object: object.id,
      likeActivity: likeActivity.id,
      undoActivity: undoActivity.id
    })
    await this.#objectStorage.removeFromCollection(
      object.id,
      'likes',
      likeActivity
    )
    await this.#objectStorage.removeFromCollection(object.id, 'likers', actor)
  }
```

Shares work exactly the same:

```
  async #handleUndoAnnounce (bot, undoActivity, shareActivity) {
    const actor = this.#getActor(undoActivity)
    shareActivity = await this.#ensureProps(
      bot,
      undoActivity,
      shareActivity,
      ['object']
    )
    let object = this.#getObject(shareActivity)
    if (!this.#formatter.isLocal(object.id)) {
      this.#logger.warn({
        msg: 'Undo activity object is not local',
        activity: undoActivity.id,
        shareActivity: shareActivity.id,
        object: object.id
      })
      return
    }
    try {
      object = await this.#objectStorage.read(object.id)
    } catch (err) {
      this.#logger.warn({
        msg: 'Share activity object not found',
        activity: undoActivity.id,
        shareActivity: shareActivity.id,
        object: object.id
      })
      return
    }
    if (!(await this.#authz.canRead(actor, object))) {
      this.#logger.warn({
        msg: 'Share activity object is not readable',
        activity: undoActivity.id,
        shareActivity: shareActivity.id,
        object: object.id
```

```
      })
      return
    }
    this.#logger.info({
      msg: 'Removing share',
      actor: actor.id,
      object: object.id,
      shareActivity: shareActivity.id,
      undoActivity: undoActivity.id
    })
    await this.#objectStorage.removeFromCollection(object.id, 'shares', shareActivity)
    await this.#objectStorage.removeFromCollection(object.id, 'sharers', actor)
  }
```

For blocks, there's not much to do, so I made that a no-op until I think up what to do with it.

Finally, for follows, just remove the actor from the `followers` collection:

```
  async #handleUndoFollow (bot, undoActivity, followActivity) {
    const actor = this.#getActor(undoActivity)
    followActivity = await this.#ensureProps(
      bot,
      undoActivity,
      followActivity,
      ['object']
    )
    const object = this.#getObject(followActivity)
    if (object.id !== this.#botId(bot)) {
      this.#logger.warn({
        msg: 'Follow activity object is not the bot',
        activity: undoActivity.id,
        object: object.id
      })
      return
    }
    if (!(await this.#actorStorage.isInCollection(
      bot.username,
      'followers'
       actor))
      ) {
      this.#logger.warn({
        msg: 'Undo follow activity from actor not in followers',
        activity: undoActivity.id,
        followActivity: followActivity.id,
        actor: actor.id
      })
      return
    }
    await this.#actorStorage.removeFromCollection(
      bot.username,
      'followers',
      actor
```

```
    )
    await this.#actorStorage.removeFromCollection(
      bot.username,
      'pendingFollowers',
      actor
    )
  }
```

That's really it for receiving remote activities. The primary work is in adding to the inbox and managing the cache, with a few activities that have side effects that you have to track. It's not that hard to be part of the fediverse!

Filtering Activities

Now that I've discussed how to properly receive and handle remote activities, I want to discuss how to ignore and discard them. That may sound kind of counterintuitive, but hear me out.

We've already discussed the `Block` activity and how it can be used to prevent bad interactions such as spam, abuse, or illegal content. But a block is entirely personal; the actor has to have a bad interaction before blocking. There are ways, at the server level, to proactively reduce those bad interactions for all actors.

In the `inbox` and `sharedInbox` handler code, an ActivityPub protocol server can examine the incoming activities and decide on one of three courses of action: silently dropping the activity; returning an error code; or delivering the activity to local actors.

It is 100% up to the server operators which of these actions the server should do (depending on the server software they have, of course). Good ActivityPub software allows configuring these filters server-wide or per user, or both.

Here are some filtering strategies that social web servers can use. Many use multiple strategies, providing additional layers of defense for users:

Keyword filtering
: One common filtering strategy is to keep a list of prohibited terms (slurs and hate words, for example) that aren't acceptable for delivery to the actor. This requires inspecting the text properties of incoming activities—at least `name`, `summary`, and `content`.

Domain name filtering
: Good server operators take responsibility for their users and take corrective action, like cancelling accounts, when those users interact badly with others. Badly maintained servers can become a safe haven for spam and abusive or harassing behavior. And some operators set up servers specifically for bad activities. Blocking incoming activities from badly moderated servers, based on domain

name, is an incredibly effective mechanism for significantly reducing bad interactions. Organizations like IFTAS provide curated domain blocklists that others can use to automatically filter incoming activities.

Naive Bayesian filtering
: Naive Bayesian filtering is a technique derived from email filters. It uses the fact that some words may not be a problem on their own, but combined with other words, in a particular context, might be spammy or abusive. Naive Bayesian filters are trained on known-good and known-bad data sets, and can be improved with feedback from users when their classifications are in error.

Other ML filtering
: Filtering based on more sophisticated machine learning (ML) algorithms is also possible. They can be applied to text content or even image, audio, or video content. However, the more sophisticated the algorithm, the more expensive the filtering is in terms of time and compute costs.

Configuring these filters, or others, is not a built-in part of the ActivityPub standard. Servers that support the `Flag` activity usually include affordances to train or configure filters based on the flagged example object. Servers may also include a *spam folder* or other collections of filtered activities; this is also not part of the ActivityPub API but it would make a great extension.

Optimizing Federated Servers

A simple server like *activitypub.bot* is going to give pretty good performance. So far, I've included some of the typical optimizations necessary for ActivityPub, like queueing and caching. I want to highlight other optimizations here:

ActivityPub API optimizations
: Chapter 3 lists a handful of good ActivityPub API optimization techniques. Most are useful for the ActivityPub API client built into your server. One that *won't* work very well with the HTTP Signatures authentication method is the HTTP caching recommendation. Because of the way HTTP Signatures manages authentication including dates, giving a different `Signature` header value each time, HTTP caching mechanisms just won't work. Using an ad hoc cache mechanism instead, as you've seen in this chapter, can do most of it for you higher up the stack.

Connection pooling
: I mentioned connection pooling briefly in Chapter 3, but it can be a much bigger deal for activity delivery over the ActivityPub protocol. A server with a lot of users might send activities to the same collection of remote servers almost continuously, so keeping those connections open saves a lot of connection setup and

teardown time. In Node.js, you're looking for the `http.Agent` class to do this work of managing connections; in Python, requests will use connection pooling if you use a `Session`.

DNS caching
: For setting up connections to new servers or connections that have dropped out of the pool, resolving hostnames can be a nontrivial part of the total time. DNS fails a lot for unpredictable reasons too, causing retries. If your operating system supports caching at the local level, great. If not, consider installing a caching DNS server for your ActivityPub server to use, or using a DNS caching library in your application.

Following a Server Checklist

As a wrap-up of Chapters 2, 3, and 4, I want to give a short checklist for building a minimal ActivityPub server. These are short summaries of the correct functionality; check the full text for more details. I've done a few of these, and in general these steps in order make the most sense, but it's also OK to scramble them around if you want to do something else first. This list makes a good starting point for a kanban board:

1. **Object and user storage**. If you have an existing storage solution and schema (say, you're integrating an existing service with ActivityPub) map the properties of AS2 objects to the fields in your database. If not, pick a mechanism from "Storing AS2 Documents" on page 62 for storing AS2 objects. This might also require making some decisions about database schema. You should at least have interfaces to read an object, create an object, update an object, delete an object, check membership of a collection, iterate through a collection, add to a collection, and remove from a collection. If you're going to keep cached remote objects separate from locally managed objects, decide how.
2. **Authorization functions**. You'll need methods to check for authorization, like the `canRead` and `canWrite` functions defined in this chapter.
3. **OAuth 2.0**. Incorporate a library for OAuth 2.0, or a server solution, as mentioned in Chapter 3. Implement server metadata, dynamic client registration, authorization, access tokens, and token renewal. Finally, implement an authorization mechanism that can turn an incoming token into an actor ID.
4. **HTTP Signatures validation**. Validate an incoming signature and turn it into an actor ID.
5. **GET object endpoint**. This is a REST API endpoint to get an AS2 object. Depending on your storage structure, you may need multiple endpoints for different kinds of objects. Use OAuth and/or HTTP Signatures to determine the

actor making the request; load the object from storage; and check whether the actor `canRead` the object. If so, return a full representation of the object.

6. **GET actor endpoint**. Depending on your storage structure, you may need a different endpoint for actors. Note that the actor should include all the required properties, and a `preferredUsername` for WebFinger.
7. **GET collection**. You'll need an endpoint for getting a collection, either with its own `items` property, or with pages. This should include filtering out members of the `items` array that the actor can't read.
8. **GET collection page**. Furthermore, you'll need an endpoint for getting a page from a collection. This should include filtering out members of the `items` array that the actor can't read.
9. **GET object collections**. As a specific requirement, implement the endpoint to get each of the `replies`, `likes`, and `shares` collections for an object.
10. **GET actor collections**. Implement endpoints for all the required actor collections (namely, `inbox`, `outbox`, `followers`, `following`, `liked`). These have different types of elements, but otherwise don't require any special handling unless your storage does.
11. **WebFinger**. Implement the endpoint at *.well-known/webfinger*. Check that the username exists, and if it does, show the actor ID. (At this point, you have a pretty complete read-only ActivityPub service. Great job so far!)
12. **GET remote objects**. Using HTTP Signatures, get a remote object by ID. A cache makes this much faster!
13. **Remote delivery**. Implement the code necessary to post a signed activity to another server. This includes deriving inboxes based on actor IDs, using shared inboxes, and signing outgoing HTTP requests. You'll also need to choose and implement a queuing strategy (in memory, persistent, etc.). Also, implement retries, with an eventual failure mode.
14. **Distribution to local actors**. Deliver activities to the inboxes of local actors. This depends on your storage structure for the `inbox` collection. This can quickly become a bottleneck, so it's a good idea to use a queueing mechanism here too, even if it seems unnecessary at a small scale.
15. **POST inbox**. Handle POST requests to an actor's `inbox` endpoint for remote activities. Validate the HTTP Signatures authentication, apply any side effects, and add the activity to the receiving actor's inbox.
16. **Incoming `Follow` side effects**. Validate the activity; then add it to a queue, like `pendingFollowers`.
17. **Incoming `Accept/Follow` side effects**. Validate the activity; then move from `pendingFollowing` to `Following`.

18. **Incoming `Reject/Follow` side effects**. Validate the activity; then remove from `pendingFollowing`.
19. **Incoming `Create` side effects**. Cache the object, including any binary. If the created object has an `inReplyTo` property that matches a local object, add the created object to the local object's `replies` collection and generate an `Add` activity.
20. **Incoming `Update` side effects**. Cache the object.
21. **Incoming `Delete` side effects**. Clear the cache for the object.
22. **Incoming `Like` side effects**. If the `object` is local, add the activity to the local object's `likes` collection.
23. **Incoming `Announce` side effects**. If the object is local, add the activity to the local object's `shares` collection.
24. **Incoming `Undo/Follow` side effects**. Remove the `actor` of the `Undo` from the `followers` of the local actor.
25. **Incoming `Undo/Like` side effects**. Remove the activity from the likes of the local object.
26. **Incoming `Undo/Announce` side effects**. Remove the activity from the `shares` of the local object.
27. **Incoming `Add` side effects**. Clear the cache for the collection and all its pages.
28. **Incoming `Remove` side effects**. Clear the cache for the collection and all its pages.
29. **Incoming `Flag` side effects**. Notify a moderator.
30. **POST outbox**. Accept a posted activity to an actor's `outbox`. Validate the OAuth 2.0 token. Give the activity an ID and timestamps. Apply any side effects. Distribute to the actor's `outbox` collection and `inbox` collection. Deliver to remote addressees.
31. **Outgoing `Create` side effects**. Store the `object` if it doesn't already have an `id`, including `likes`, `replies`, and `shares` collections and timestamps. If the object has an `inReplyTo` value that is a local object, add the created object to the local object's `replies` collection and generate an `Add` activity.
32. **Outgoing `Update` side effects**. Validate that the actor can write the object. Store the changed properties in the `object`, making sure not to overwrite any server-managed properties like ID, collections, and timestamps.
33. **Outgoing `Delete` side effects**. Validate that the actor can write the object. Replace the object with a `Tombstone`, including `formerType`.
34. **Outgoing `Follow` side effects**. Add the activity to the `pendingFollowing` collection. If the object is a local actor, add the activity to their `pendingFollowers` collection.

35. **Outgoing `Accept/Follow` side effects**. Remove the activity from the `pendingFollowers` collection. Add the object to the `followers` collection. If the follower is a local actor, add the actor to their `following` collection.
36. **Outgoing `Reject/Follow` side effects**. Remove the activity from the `pendingFollowers` collection. If the follower is a local actor, remove the activity from their `pendingFollowing` collection.
37. **Outgoing `Block` side effects**. Remove the `object` from `followers` and `following`. Add the object to the `blocked` collection. If the `object` is a local actor, remove the blocking actor from their `following` and `followers`. Do not deliver the activity to the `object`.
38. **Outgoing `Like` side effects**. Add the object to the actor's `liked` collection. If the object is local, add the activity to the object's `likes` collection.
39. **Outgoing `Announce` side effects**. If the object is local, add the activity to the object's `shares` collection.
40. **Outgoing `Undo/Follow` side effects**. Remove the object from the actor's `following` collection. If the object is a local actor, remove the actor from the local actor's `followers` collection.
41. **Outgoing `Undo/Like` side effects**. Remove the object from the actor's `liked` collection. If the object is local, remove the activity from the object's `likes` collection.
42. **Outgoing `Undo/Block` side effects**. Remove the object from the actor's `blocked` collection.
43. **Outgoing `Undo/Announce` side effects**. If the object is local, remove the activity from the object's `shares` collection.
44. **Outgoing `Add` side effects**. If the `target` is local, add the `object` to the `target` collection.
45. **Outgoing `Remove` side effects**. If the `target` is local, remove the `object` from the `target` collection.
46. **Outgoing `Flag` side effects**. If the `object` is local, notify a moderator.
47. **File upload**. Upload a file and distribute the corresponding `Create` activity.
48. **`proxyUrl`**. Request objects remotely and return them to the client. Cache objects for faster response later. Check authorization.
49. **POST `sharedInbox`**. As an optimization, implement the `sharedInbox` endpoint. Distribute to local inboxes based on addressing, including membership in addressed collections, like a `followers` collection.

This should get a pretty functional ActivityPub server working, with all the basic activity types covered. After this point, incorporating optimizations and adding extensions becomes easier.

Conclusion

Are you feeling a little underwhelmed? I hope so.

Actually getting to the ActivityPub protocol can feel somewhat mundane. But consider how much you've gone through to get to this point: understanding AS2, the ActivityPub API, and all the extra bits and pieces necessary to make the whole network work.

The activities I've covered so far are the basics you need to make an ActivityPub server run, but they're not all that you can do. In Chapter 5, I'll explain how to use ActivityPub extensions to do all sorts of cool and surprising things with this infrastructure.

CHAPTER 5
Extending ActivityPub

At its base specification, ActivityPub covers some of the most essential activities that we do with social networks: creating and editing content, reacting to content, organizing content, and managing the social graph. For many social applications, this is going to cover most of what you need.

But human sociality is large and sprawling, and we can use ActivityPub for thousands of social applications. The goal of ActivityPub is to be a *platform* for building social applications, not to be the endpoint for a tiny subset of the social world. If a form of social interaction can be mediated or represented digitally, ActivityPub should be able to handle it. That's why ActivityPub is *extensible*.

Extensibility is built into the ActivityPub model because it's built into the Activity Streams 2.0 model, which (in turn) is built on top of JSON-LD, as you learned in Chapter 1. Adding new properties and even new data types can facilitate different types of social interaction online.

In this chapter, I'm first going to talk about general principles for *receiving* extended properties or object types that your program wasn't designed for. Then I'll cover the flip side: *sending* properties and object types that other programs might not be able to understand. I'll be using made-up extensions to illustrate these principles.

I'll then get into two major categories of extended types: using other parts of the Activity Vocabulary and using other well-known vocabularies. Finally, I'll talk about developing a new vocabulary from scratch—how to design it, how to implement it, and how to get others around the social web to use it too.

I'd like to reemphasize that the examples in the following few sections are for *fictional, nonexistent properties and types* at the time of this writing in 2024. I tried to choose examples that would be useful for social-network applications, though, so it's entirely possible that by the time you read this, some of these terms will have been

defined and may even be in wide use. I can say that unless the internet has gone *very* sideways in your timeline, none of the extensions you see in the real world will use context documents with domain names in the *.example* top-level domain. Those fictional extensions will stay fictional, in implementation if not in spirit.

Understanding Senders and Receivers

Figure 5-1 is a sequence diagram for the main interactions between components in the ActivityPub network. Paying attention to the direction of data flow, you can see that three main components have to handle incoming Activity Streams objects from other components: the *sending server*, the *receiving server*, and the *receiving client*. Similarly, three components can originate activities or pass activities forward to other components: the *sending client,* the *sending server*, and the *receiving server*.

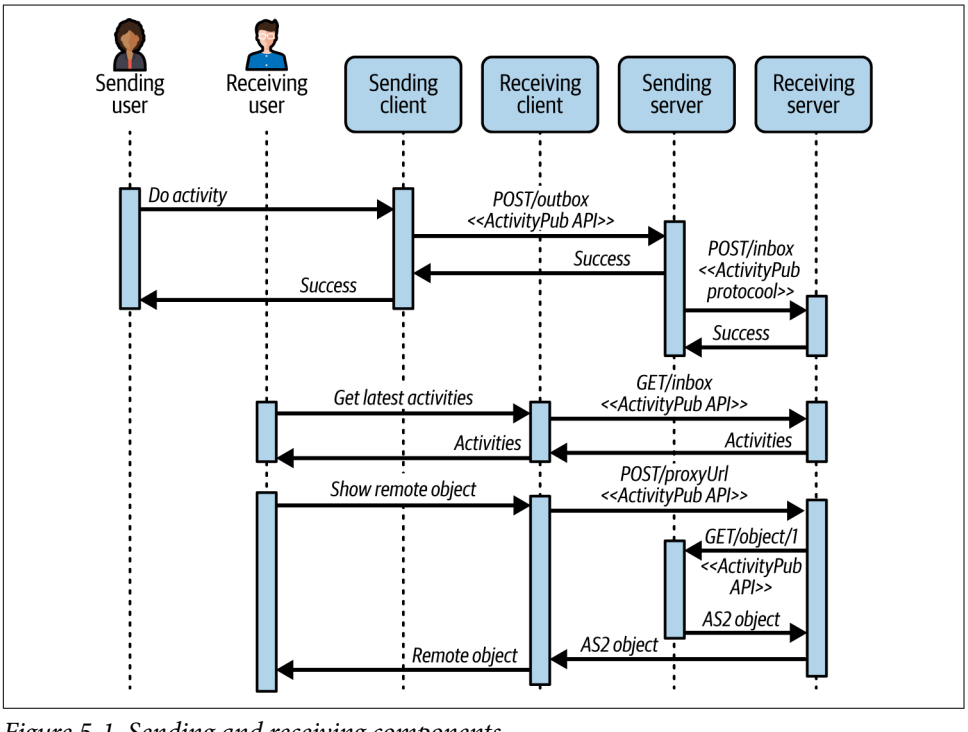

Figure 5-1. Sending and receiving components

Note that most ActivityPub client software is actually two-way, generating and receiving activities. And most ActivityPub server software is also able to send and receive activities. I've just broken these into two roles to make understanding the flow of data easier, and how extension properties and types can be handled.

The client software also has to "receive" activities from the user with whatever user interface is provided in the client. If those user interfaces have interactions that aren't described in the ActivityPub spec, you can think of them as extensions too. Similarly, the receiving client software somehow has to represent the extension data it receives, and that's another "sending" structure. I won't get too far into how that can work, but it's worthwhile recognizing that providing interesting and novel functionality to end users is the whole point of making extensions.

In the following sections, I'll talk about each of these roles—sending client, sending server, receiving server, and receiving client—and best practices for each one in handling incoming extensions and generating outgoing extensions.

Receiving Extended Properties

The simplest step into the world of ActivityPub extensions is receiving extended properties. Let's say your app has implemented the ActivityPub API, and a user posts this activity to their own outbox:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "mood": "https://extension.example/mood" }
  ],
  "to": "as:Public",
  "type": "Create",
  "object": {
    "type": "Note",
    "content": "Hello, World!"
  },
  "mood:currentMood": "mood:Friendly"
}
```

Most of this should be familiar territory by now: a `Create` activity for a `Note` object with a classic universal greeting. But the code has an added `mood:currentMood` property—in this example, representing the feeling or mood the creator had while making the note. (As of this writing, a common extension for this doesn't exist, so I'm using an example namespace.) As you may remember from Chapter 2, the external vocabulary for moods is given in the `@context` property, and we use the `mood:` prefix to distinguish it from other vocabularies.

As humans reading this JSON, you and I know that's what it is. But by definition in this segment, your ActivityPub API server wasn't built to recognize it or use it. So what should it do with the activity? The main answer is to treat it as usual, without any special processing. Really, if you think about it, the basic processing of a `Create` activity is going to be exactly the same with or without the `mood` property: the API server will create the object, store it on the server, put it in the actor's own inbox and outbox, and deliver the activity to all the actor's followers.

However, remember that activities have a long lifetime and a broad distribution. This activity will appear in the actor's inbox, their outbox, and in the inboxes of all the actor's followers. They may have ActivityPub client software that is aware of and will process the `mood:currentMood` property—for example, by showing a smiling emoji or a waving hand. As an ActivityPub API server, your software shouldn't ruin that experience for your users or people elsewhere on the fediverse.

For API servers that use AS2 as their native storage format, keeping extension properties is not that hard; in fact, you'd have to write extra code to scrape out properties that weren't recognized. But for servers that use other storage formats (say, existing social applications that have added AP support), retaining those properties can be trickier. Often these applications will map properties they recognize, like `actor` and `content`, onto fields already found in their database. If they didn't plan ahead for the `mood:currentMood` property, it can get dropped off.

You can deal with this in a few ways. Some content management system (CMS) servers and other big server codebases have data structures that allow additional name-value property pairs for objects in the database. Saving the activity's extra properties there can help keep them around. Another option is to add a database field and stick a JSON representation of the activity's extra properties into it, separate from your application's existing database tables. Regardless, finding options for storing unrecognized properties and passing them on to ActivityPub API clients (or other ActivityPub federation protocol servers) makes your software a better part of the fediverse.

For ActivityPub protocol servers, the behavior will be somewhat similar when they receive an activity with an unrecognized extension property from a remote originating server. The rest of the activity can be processed as usual and stored in the recipients' inboxes. If the activity type has side effects, they can be implemented as normal. It's helpful to retain the property to deliver to ActivityPub clients for their actors, but otherwise, no additional work needs to be done.

For ActivityPub API clients, receiving an unrecognized extension property shouldn't require additional work either. Preserving the property isn't even necessary, unless you want to keep it around for debugging purposes. If your client software doesn't know how to represent it in a meaningful way for your user, it might as well not exist.

One point to note is that correctly parsing and interacting with AS2 objects with extended properties requires a JSON-LD-aware parser. For a lot of ActivityPub processing, you can use a regular JSON parser and be more or less confident that your software understands the incoming content; that was part of the design of AS2. But once there are extensions, with namespaces, prefixes, and other tricky bits, using JSON-LD natively is important. You can find a list of good JSON-LD parsing libraries for all kinds of programming languages at the JSON-LD website.

Receiving Extended Types

Extended types are similar to extended properties, but with a little more complication. You can use extended types in ActivityPub in two ways: as properties of an activity and as activities themselves.

Let's say an ActivityPub API server receives this example activity, which comes from a fictional farm-simulation game:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "farm": "https://farmgame.example/ns/" }
  ],
  "to": "as:Public",
  "type": ["farm:Plant", "Create"],
  "summary": "Evan planted broccoli on the northern plot of his farm",
  "object": {
    "id": "https://farmgame.example/crop/17504",
    "type": ["farm:Crop", "Object"],
    "summary": "A broccoli crop",
    "icon": {
      "type": "Link",
      "mediaType": "image/png",
      "height": 512,
      "width": 512,
      "url": "https://farmgame.example/images/field-of-broccoli.png"
    }
  },
  "farm:food": "farm:Broccoli",
  "location": {
    "type": ["farm:Plot", "Place"],
    "name": "Northern plot",
    "id": "https://farmgame.example/plot/2635"
  }
}
```

This activity has an extended type, and two of its properties also have extended types. The code is supposed to represent an activity where the user has planted a fictional crop in a plot on their fictional farm. The `Plant`, `Crop`, and `Plot` types are new; maybe this game was created after the ActivityPub API server was even built. What should the server do with this activity?

The good news is that the client application (the farm-simulation game) has provided fallback types from the Activity Vocabulary. The ActivityPub API server should probably know how to handle that; `Create` is one of the core activity types in ActivityPub, after all. So the server can give the activity an ID value, add the `actor`, add a publication timestamp, cache the `object` and `location` information if it wants to, and then go about distributing this activity to the user's followers around the social web—doing its best to preserve property values as they're distributed.

For the ActivityPub protocol server on the receiving end, the process is almost the same. As you remember from Chapter 4, fewer side effects usually occur on the receiving server's side. It might do some caching and distribute the activity to the inboxes of the receiving actors, but otherwise this `Plant` activity will be treated about the same as any other activity.

The biggest burden is on the receiving ActivityPub client application. This is an unknown activity type, so the application won't have built-in representations for types like `Plant`, `Crop`, and `Plot`. However, all AS2 objects, including activities, have properties that provide fallback representations for the object in text and graphics. With the preceding activity, the app can use the `summary` for a high-level description of the activity (say, in a list of notifications). If the app needs a more complex representation, it could build one from the `name` and `icon` of the `object`, whatever additional information is provided about the `actor`, as well as timestamps and links.

What if nothing in the activity gives any further explanation? That's OK too. The application can fall back to a generic description (*Evan did…something? Apparently?*) or just ignore the activity entirely. It's great if an ActivityPub client can represent interesting new kinds of activities with fallback representations in standard properties, but in their absence, it's OK to just skip over the activity or provide a minimal marker.

Sending Extended Properties

Let's reverse the flow here and talk about *sending* activities with extended properties. If you've read through the previous two sections, you'll realize that one of the most important rules of extended properties is that they have to be *optional* and *backward compatible*. Many (maybe most) ActivityPub software processors aren't going to have built-in support for your extended property. You can't count on other processors to change their default behavior because of your new property.

Take, for example, a movie-review client that posts this activity to an ActivityPub API server:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "spoilers": "https://spoilers.example/ns/" }
  ],
  "to": "as:Public",
  "type": "Create",
  "object": {
    "type": "Note",
    "content": "In this movie, 'Rosebud' was his sled.",
    "spoilers:summaryContainsSpoilers": true,
    "tag": {
      "id": "https://movies.example/1941/citizen-kane",
```

```
        "type": "Video",
        "name": "Citizen Kane (1941)"
      }
    }
  }
```

The intent here is that client software would hide the `content` unless the user agrees to the plot being spoiled. But since most software won't be aware of this UI requirement, it will blithely go ahead and spoil this 80-year-old film plot. (Sorry, reader. It's a great film even if you know this plot point, I promise!) The `sensitive` flag, discussed in "Miscellany" on page 199, functions approximately the same way, although it is well supported on the social web.

With a little creativity, though, it's possible to design extension structures that work even if the receiving software doesn't know how to process the extended properties. Here's a better structure for the plot-spoiler property:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "spoilers": "https://spoilers.example/ns/" }
  ],
  "to": "as:Public",
  "type": "Create",
  "object": {
    "type": "Note",
    "content": "At the end we find out what Rosebud was.",
    "spoilers:spoilers": "It was his sled!",
    "tag": {
      "id": "https://movies.example/1941/citizen-kane",
      "type": "Video",
      "name": "Citizen Kane (1941)"
    }
  }
}
```

Now, the actual spoilers have been moved to the separate property, which will be shown by only compliant processors. Other readers will not have the plot spoiled at all, but they also won't get to see the spoilers even if they want to! We're using the fact that extension properties will be invisible on clients that don't recognize them to prevent people from accidentally seeing the spoilers.

ActivityPub API servers and ActivityPub protocol recipients are *mostly* going to pass through properties that they don't recognize. Therefore, properties intended to restrict or modify the way this social infrastructure works are unlikely to be useful. But properties that could *limit* or *guide* processing can give hints that optimize the process. Here's an activity, as posted to an ActivityPub API server, with an example extension property for noting its relevance:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "relevance": "https://relevance.example/ns/" }
  ],
  "to": "as:Public",
  "type": "Create",
  "object": {
    "content": "My band is playing on Feb 21 at 7PM.",
    "relevance:notAfter": "2024-02-21T21:00:00Z"
  }
}
```

A compliant processor could use this hint to, say, stop trying to deliver the activity after its object is no longer relevant. Processors that don't recognize the property won't benefit from the optimization, but they at least won't do a *worse* job.

ActivityPub servers can add extension properties of their own. For example, a sending server might add the number of attempts it has made before successfully completing delivery. A receiving server could add a `received` timestamp. These could be useful for clients to explain delivery delays or for administrative debugging.

One important way that ActivityPub servers add extended properties is through the read-only API. Take, for example, an ActivityPub server that tracks the `View` activities that actors post for an object. The `View` activity type, in the Activity Vocabulary, is used to show that an actor has viewed an object, like a video or image. When an ActivityPub API client sends a `GET` request for the object, the server could include a (fictional) `views` property on the object to share the collection of activities:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://namespace.example/ns/views"
  ],
  "attributedTo": "https://social.example/users/189",
  "to": "as:Public",
  "type": "Video",
  "id": "https://social.example/video/2370",
  "name": "Me solving the Rubik's cube",
  "views": {
    "id": "https://social.example/video/2370/views",
    "type": "OrderedCollection",
    "totalItems": 26694
  }
}
```

The `views` property, here, is defined in the additional context document used in the `@context` property. It doesn't interfere with the other parts of the `Video` object; a client application that was not programmed to understand the `views` property can continue processing here without any problems.

A useful example of extended properties is for ActivityPub actor objects. Let's say that an ActivityPub server maintains a `karma` score for each actor, calculated from the number of original links they've shared, or likes they've received, or comments they've posted. The actor document might look something like this:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://namespace.example/ns/karma"
  ],
  "type": "Person",
  "id": "https://social.example/user/kfury",
  "name": "Kevin F.",
  "inbox": "https://social.example/user/kfury/inbox",
  "outbox": "https://social.example/user/kfury/outbox",
  "followers": "https://social.example/user/kfury/followers",
  "following": "https://social.example/user/kfury/following",
  "likes": "https://social.example/user/kfury/likes",
  "karma": 81322
}
```

Here, the `karma` property has been added to show the actor's karma score. An ActivityPub API client app or an ActivityPub protocol server could use this information for making decisions about this actor's posts, or the user could just show it off as a point of pride!

Another great use case for extensions in actor documents is to optimize some of the more tedious parts of the API for client developers. For example, consider an ActivityPub API client that wants to show the user a list of their *mutual* connections—actors they follow who also follow them back. In unidirectional social-graph models like the ActivityPub architecture, this is usually a good approximation for your "real friends."

There's no easy way for an ActivityPub API client application to get this list of mutual connections. AP actors have a `followers` collection for actors that follow them and a `following` collection for actors they follow. A naive (and somewhat exhausting) implementation would be to fetch *all* the actors from the `followers` collection and *all* the actors from the `following` collection, and then take the intersection to find the list of mutuals. That would be terrible!

Although this data is difficult for client apps to get through the standard API, it's probably not that difficult for the server to calculate, since it's keeping track of the graph already. An ActivityPub API server could offer a `mutuals` collection for the actor, to help client apps quickly calculate and show this information:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://namespace.example/ns/mutuals"
```

```
  ],
  "type": "Person",
  "id": "https://social.example/user/mattl",
  "name": "Matt L.",
  "inbox": "https://social.example/user/mattl/inbox",
  "outbox": "https://social.example/user/mattl/outbox",
  "followers": "https://social.example/user/mattl/followers",
  "following": "https://social.example/user/mattl/following",
  "mutuals": "https://social.example/user/mattl/mutuals",
  "likes": "https://social.example/user/mattl/likes"
}
```

ActivityPub client apps can use this optimization to speed up calculating the list of mutuals for servers that provide it, and fall back to the slower and more difficult method for servers that don't.

Extending actors and objects in this way has a few downsides. One is bandwidth: if an actor has twice as many properties, it's going to have about double the size for downloads. The other is complexity: an actor with 50 or 100 properties from dozens of extensions lacks the same conceptual cohesion of a simple `Person` object with only a few closely related properties.

Sending Extended Types

The most interesting way to extend ActivityPub is to use extended activity and object types. This mechanism lets us not just enhance existing interactions but develop entirely new ones.

Say, for example, you want to implement a feature in an ActivityPub client to let a person greet or acknowledge one of their contacts wordlessly, as a nice way to tell people you care about that you're thinking of them, without requiring any response. Facebook has a "poke" feature for this kind of interaction; other social networks use "wave" or "ping" instead.

Here's what a fictional example `Ping` activity might look like when an ActivityPub API client posts it to the API server:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://namespace.example/ns/ping"
  ],
  "type": ["Ping", "Activity"],
  "summaryMap": {
    "en": "Evan pinged Chris"
  },
  "to": ["https://social.example/users/blizzard"],
  "object": "https://social.example/users/blizzard"
}
```

By now, this extension structure should be familiar. An additional line in the `@context` property includes the context for the (again, fictional) ping protocol. The `type` property includes the `Ping` type, but also includes the closest related type from the Activity Vocabulary. This will often just be `Activity` or `IntransitiveActivity`. Finally, the addressing property tells the server where to route the activity.

Well-behaved ActivityPub API servers will do some default processing on this activity—adding an `id` for the activity, including the `actor` property, and adding `published` and `updated` timestamps. The server doesn't know anything about the `Ping` interaction we're using, but it knows how to create, store, and route activities. It should make sure the activity gets delivered to *social.example* for user *blizzard*.

The receiving ActivityPub protocol server should, on its part, make sure the activity gets routed to the inbox for user *blizzard*. From there, the user's ActivityPub API client can take over.

Clients on the receiving end, when they see a `Ping` activity in the user's inbox, are free to interact in whatever way that makes sense. A web client might show a pop-up notification or keep a running list of the latest `Ping` activities by that sender. They might include some UI elements to send back an acknowledgment—a `Pong` activity, say.

Receiving and processing a single `Ping` activity should be straightforward, but collecting all of one actor's activities of a particular type is difficult. The interface for getting new activities is narrow. With the default ActivityPub API interface, the only way to find out about new incoming activities is through the actor's `inbox` collection, which is a linear, reverse-chronological list of all activities the actor has ever received. Searching this collection for activities of a particular type is time-consuming and uses a lot of bandwidth. This is a great opportunity for the kind of server-side optimization I discussed in the previous section—a `pings` collection for the actor that lists only the most recent `Ping` activity from each contact.

The `Ping` activity doesn't have any persistent state; it just travels through the pipes of the ActivityPub network from one actor to another. However, an interaction that requires state management will typically require an ActivityPub API client that can host ActivityPub objects on the web and update their state.

Let's try out a social interaction that's common in the real world—a group of people signing a birthday card for a friend or coworker. It might have an object type like `BirthdayCard`, an `Invite` activity to invite someone to sign the card, a `Sign` activity for signing the card, and then an `Announce` activity to deliver the card to the recipient.

A first person could create the birthday card by using some kind of client web app on *birthdaycard.example* that sends AS2 activities to their AP API server on *activitypub.example*:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://namespace.example/ns/birthdaycard"
  ],
  "type": "Create",
  "object": {
    "id": "https://birthdaycard.example/id/26056",
    "type": ["BirthdayCard", "Object"],
    "cardFor": "https://social.example/user/lprodrom"
  }
}
```

As you saw in Chapter 3, if an object already has an ID, it doesn't need to be created on the ActivityPub API server. Note also that, by default it doesn't have any addressees. The next step is inviting other people to sign the card:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://namespace.example/ns/birthdaycard"
  ],
  "type": "Invite",
  "object": {
    "id": "https://birthdaycard.example/id/26056",
    "type": ["BirthdayCard", "Object"],
    "cardFor": "https://social.example/user/lprodrom"
  },
  "target": [
    "https://social.example/user/andyp",
    "https://social.example/user/ted",
    "https://social.example/user/nate"
  ]
}
```

This is where the process gets tricky. Ideally, once the other people are invited, they should at least be able to read the card.

But how will the server at *birthdaycard.example* know that the `Invite` activity has been sent, in order to change the access control on the `BirthdayCard` object? Mostly, the *birthdaycard.example* service must be the one generating the invitation, maybe with an integrated web or mobile UI.

Each of the signers can send a `Sign` activity:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://namespace.example/ns/birthdaycard"
  ],
  "type": ["Sign", "Activity"],
  "to": "https://activitypub.example/user/evan",
```

```
  "object": "https://birthdaycard.example/id/26056",
  "content": "Happy birthday, Lisa!"
}
```

Note that the `Sign` activity is directed to the original sender of the invitation. This is usually the best way to manage protocols, although the *birthdaycard.example* client app will need to find these activities in the organizer's inbox, which is kind of a hassle. Another possibility is making the `BirthdayCard` object a full ActivityPub actor, so it has its own inbox and can receive activities directly. The design is dependent on the problem domain; for this imaginary interaction, I'm going to use the sender-inbox method.

Finally, the original sender can send an `Announce` activity to share the birthday card with the intended recipient. Surprise!

The `Announce` activity might look something like this:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://namespace.example/ns/birthdaycard"
  ],
  "type": "Announce",
  "object": {
    "id": "https://birthdaycard.example/id/26056",
    "type": ["BirthdayCard", "Object"],
    "cardFor": "https://social.example/user/lprodrom",
    "signatures": {
      "type": "Collection",
      "id": "https://birthdaycard.example/id/26056",
      "items": [
        "https://social.example/user/andyp/sign/33",
        "https://social.example/user/ted/sign/14",
        "https://social.example/user/nate/sign/327"
      ]
    }
  },
  "to": "https://social.example/user/lprodrom",
  "cc": [
    "https://social.example/user/andyp",
    "https://social.example/user/ted",
    "https://social.example/user/nate"
  ]
}
```

Here, the card is shared to the recipient and all the signatories. Note that the app has kept a collection of `Sign` activities in the `signatures` property.

You can see a lot of interesting territory here. Nothing is stopping an ActivityPub API server from incorporating this kind of interaction too, but it would be largely transparent to other actors on the fediverse. Everyone else would just see ActivityPub

objects being created, shared, and modified by ActivityPub activities—some with standard types, others with extended types.

So, that's a lot of discussion on how to properly receive and send extension properties and types. The key principles are these: your software should do its best to pass through extension properties and types it doesn't understand. And if your software sends extension properties and types, they should be designed to work properly even if the intermediate software components pass them through but don't understand them.

I've just given you pages and pages of plausible-sounding extensions that do interesting social interactions, and I've continually told you that they're not real and you can't use them on the fediverse today. Let's get into some of the types and properties you *can* use today and show how to define your own extensions to do even more.

Using the Rest of the Activity Vocabulary

The ActivityPub API and federation protocol use a relatively small subset of the total Activity Vocabulary: `Create`, `Update`, and `Delete` for content; `Add` and `Remove` for collections; `Follow`, `Accept`, `Block`, and `Reject` for managing the social graph; `Announce` and `Like` for reactions; and `Undo` to clean up mistakes.

But the Activity Vocabulary includes 20-plus more activity types that you can use out-of-the-box with ActivityPub (see Appendix A for the full list). However, not all ActivityPub servers and clients will recognize them and implement side effects; most will treat them like any other extended activity. They break down according to a few interesting subgroups, which I document in the subsections that follow.

Unlike other extensions, these don't need any change to the `@context` property, since they're part of the AS2 vocabulary. That's the upside. The downside is that the Activity Vocabulary types are sometimes too sparse to use for detailed interactions on the social web. I'll note where the types are a little too tight for my liking, with the hope that you can supplement them with other well-known vocabularies (or even build your own).

Polls

One popular activity type on the fediverse is `Question`, which represents asking a question—it's a verb, not a noun! There are two main kinds of questions.

One is an *open-ended* question, like you'd see on Quora or Stack Overflow. The `replies` collection lists all the answers given, and the `result` value can be the selected answer. Here's what one may look like:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Question",
```

```
  "actor": "https://social.example/users/evanp",
  "id": "https://social.example/users/evanp/question/328",
  "content": "Where can I get good BBQ in Austin?",
  "replies": {
    "type": "OrderedCollection",
    "id": "https://social.example/users/evanp/question/328/replies",
    "items": [
      {
        "id": "https://remote.example/users/todb/notes/1",
        "type": "Note",
        "attributedTo": "https://remote.example/users/todb",
        "content": "Micklethwait Craft Meats has great ribs and brisket."
      }
    ]
  },
  "result": "https://remote.example/users/todb/notes/1"
}
```

The second, more structured form with multiple-choice questions uses the `oneOf` property to define acceptable answers:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Question",
  "actor": "https://social.example/users/evanp",
  "id": "https://social.example/users/evanp/question/329",
  "content": "What should I eat at the BBQ joint?",
  "oneOf": [
    {
      "id": "https://social.example/users/evanp/question/329/option/1",
      "type": "Note",
      "name": "Brisket"
    },
    {
      "id": "https://social.example/users/evanp/question/329/option/2",
      "type": "Note",
      "name": "Pork ribs"
    }
  ],
  "replies": {
    "type": "OrderedCollection",
    "id": "https://social.example/users/evanp/question/329/replies",
    "items": [
      {
        "id": "https://remote.example/users/todb/notes/2",
        "type": "Note",
        "attributedTo": "https://remote.example/users/todb",
        "inReplyTo": "https://social.example/users/evanp/question/329",
        "name": "Brisket"
      }
    ]
  }
}
```

The `replies` are to the `Question`. (Mastodon includes a `replies` property on each answer `Note`; this is technically incorrect but does have the advantage of quickly summing up totals without requiring a scan of the full `replies` property of the `Question`.) They are matched to the available options by the `name` property, which is inexact but seems to work. To allow multiple answers, use `anyOf` instead of `oneOf`.

This structure works well even if the API server doesn't know how to implement side effects for questions; client apps can mostly handle posting and replying to the question, as long as the API server can manage the `replies` collection. Many servers have built-in support for this type, though, because it's so popular. It's probably the safest type to use outside of the ActivityPub spec types.

Account Portability

Another common type on the fediverse is used for moving from one account to another and letting followers automatically re-follow the new account. With two accounts, the old actor can send a `Move` activity with these properties:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://old.example/users/username/move/1",
  "to": "https://old.example/users/username/followers",
  "type": "Move",
  "actor": "https://old.example/users/username",
  "object": "https://old.example/users/username",
  "target": "https://new.example/users/username"
}
```

The new actor must have an `alsoKnownAs` property that includes the `id` of the old account. If this condition is met, when the old account's followers receive the `Move` activity, they'll send a `Follow` activity to the new account and an `Undo Follow` to the old account. This lets users quickly move from one account to another—say, when changing social-network providers or maybe changing jobs or schools.

Many ActivityPub servers support this activity type, and ActivityPub clients can also easily use it. It's not difficult to send. Recipients can automatically send the `Follow` activity without requiring server support. The actor at the old address will often have an added `movedTo` property that matches the new account address, to help with later reads.

As an aside, `Move` is also defined for a one-step move from one collection to another, like a combined `Add` and `Remove`. However, this usage is much less well implemented.

Events

Event management is common in social networking. It includes creating events, sending invitations, and collecting RSVPs. The Activity Vocabulary includes a few interesting types for event management.

Here's how to create a new `Event`:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evan/create/4744",
  "type": "Create",
  "actor": "https://social.example/users/evan",
  "object": {
    "id": "https://social.example/users/evan/event/8741",
    "type": "Event",
    "name": "Picnic dinner in Parc Laurier",
    "summaryMap": {
      "en": "Please join us for a potluck supper in the park."
    },
    "startTime": "2024-06-01T23:00:00Z",
    "endTime": "2024-06-02T02:00:00Z",
    "location": {
      "id": "https://osm.example/way/13687084",
      "nameMap": {
        "en": "Sir Wilfrid Laurier Park"
      },
      "lat": 45.53243,
      "lon": -73.58706
    }
  }
}
```

Important properties for an event include the start time, end time, and location. The creator can invite guests to the event:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/users/evan/invite/25846",
  "type": "Invite",
  "to": [
    "https://remote.example/users/tara",
    "https://further.example/users/lucinda"
  ],
  "actor": "https://social.example/users/evan",
  "object": "https://social.example/users/evan/event/8741",
  "target": [
    "https://remote.example/users/tara",
    "https://further.example/users/lucinda"
  ]
}
```

Note the order of the properties: the invitees are in the `target` property, and the event they're invited to is in the `object` property.

The invitees can accept the invitation:

```
{
    "@context": "https://www.w3.org/ns/activitystreams",
    "id": "https://remote.example/users/tara/accept/233352",
    "actor": "https://remote.example/users/tara",
    "to": "https://social.example/users/evan",
    "type": "Accept",
    "object": "https://social.example/users/evan/invite/25846"
}
```

The `object` property is the `Invite` activity, not the `Event`. An invitee can also `Reject` the invitation:

```
{
    "@context": "https://www.w3.org/ns/activitystreams",
    "id": "https://further.example/users/lucinda/reject/28061",
    "actor": "https://further.example/users/lucinda",
    "to": "https://social.example/users/evan",
    "type": "Reject",
    "object": "https://social.example/users/evan/invite/25846"
}
```

In addition, `TentativeAccept` and `TentativeReject` activity types are similar to their more emphatic counterparts.

This is a fairly simple social-event-management protocol, but it's probably just robust enough to work with public events. One complicated part is the authorization model: ideally, inviting new attendees would give them read access to the event, adding them to the `to` property of the object:

```
{
  "id": "https://social.example/users/evan/event/8741",
  "type": "Event",
  "name": "Picnic dinner in Parc Laurier",
  "summaryMap": {
    "en": "Please join us for a potluck supper in the park."
  },
  "to": [
    "https://remote.example/users/tara",
    "https://further.example/users/lucinda"
  ]
}
```

If the ActivityPub server doesn't automatically implement this change, an ActivityPub API client can use the `Update` activity to directly change the access.

Another notable absence is collection properties for all invitees, or for those who've accepted, rejected, or tentatively done either. If they reply to the event with questions,

such as about what to bring or where to post photos after the fact, it's not clear to whom the replies should be addressed. Having collections for all invitees would make that easier.

Groups

Social networks often support groups for discussions between opt-in collections of users. The group schema in AS2 is sparse, but it can be the basis for a fuller-featured group mechanism.

A person can create a group for a discussion:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "to": "Public",
  "actor": "https://social.example/users/tom",
  "object": {
    "id": "https://groups.example/group/native-plants",
    "name": "Native Plants",
    "summary": "A group for gardeners who love native plants."
  }
}
```

Groups can go through the whole CRUD cycle, including updates and deletes. The important activity types for groups are `Join` and `Leave`. A person can `Join` a group:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Join",
  "to": "https://groups.example/group/native-plants",
  "actor": "https://social.example/users/evanp",
  "object": "https://groups.example/group/native-plants"
}
```

Similarly, they can `Leave` a group:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Leave",
  "to": "https://groups.example/group/native-plants",
  "actor": "https://social.example/users/evanp",
  "object": "https://groups.example/group/native-plants"
}
```

Finally, when a user sends content to a group, the group can repost that content:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "to": "https://groups.example/group/native-plants",
  "actor": "https://social.example/users/evanp",
  "object": {
```

```
    "id": "https://social.example/users/evanp",
    "type": "Note",
    "content": "Hello everyone!"
  }
}
```

Unfortunately, the semantics of routing content to groups isn't built into ActivityPub, so servers probably need to internally support routing for groups. It's not possible to manage it at the client level like other extended vocabulary.

It's also a little frustrating that, as of this writing, groups don't have a lot of clues built in about their members or admins and whether membership approval is automatic or manual, nor do they offer any collections for shared documents, images, and so on. I think groups are essential to the social media experience, so I hope we see some extensions that flesh out what makes them so useful.

Geosocial

Geosocial activities, sometimes called *check-ins*, document an actor's movements through physical space. The `Place` object type (covered in Chapter 2) is the cornerstone of this schema, but a few interesting activity types are associated with the geosocial domain.

The most useful is `Arrive`, which shows that an actor has arrived at the place:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "to": "https://social.example/users/evanp/followers",
  "type": "Arrive",
  "actor": "https://social.example/users/evanp",
  "location": {
    "id": "https://osm.example/way/13687084",
    "type": "Place",
    "name": "Parc Laurier"
  },
  "context": "https://social.example/users/evan/event/8741",
  "summary": "Evan arrived at Parc Laurier for the picnic dinner event"
}
```

Note that the place is in the `location` property, not the `object` property.

The `Leave` activity type, conversely, shows when someone leaves a place. Many privacy-conscious people prefer this style of check-in because it doesn't show your current location:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "to": "https://social.example/users/evanp/followers",
  "type": "Leave",
  "actor": "https://social.example/users/evanp",
  "object": {
```

```
    "id": "https://places.example/us/nj/princeton/nassau-inn",
    "type": "Place",
    "name": "Nassau Inn"
  },
  "summary": "Evan left the Nassau Inn"
}
```

These two types aren't quite symmetrical; the `Leave` activity uses the `object` property to track the `Place`.

Finally, the `Travel` activity combines these two activities into one, concentrating more on the journey between locations than on the final state:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "to": "https://social.example/users/evanp/followers",
  "type": "Travel",
  "actor": "https://social.example/users/evanp",
  "origin": {
    "id": "https://places.example/ca/qc/montreal",
    "type": "Place",
    "name": "Montreal, QC"
  },
  "target": {
    "id": "https://places.example/us/nj/princeton",
    "type": "Place",
    "name": "Princeton, NJ"
  },
  "summary": "Evan traveled from Montreal, QC to Princeton, NJ"
}
```

Because these activities are relatively self-contained and don't require a change of state on the server side, they are pretty tractable for use on the ActivityPub network. The big drawback is that we don't yet have a well-known vocabulary of place IDs with Activity Streams representations on the server. I'd love to see these developed or adapted from Foursquare, Google Maps, or OpenStreetMap.

Media Experiences

Another common social-network interaction is sharing media experiences—content you watch, listen to, or read. The Activity Vocabulary has several activity types for media experiences. `Read`, for example, shows that the user has read a text, like an article, a book, or a web page:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Read",
  "to": "Public",
  "actor": "https://social.example/users/evanp",
  "object": {
    "type": "Article",
```

```
    "url": "https://magazine.example/articles/activity-pub-is-great",
    "name": "ActivityPub Is Great"
  },
  "result": {
    "id": "https://social/example/users/evanp/note/3",
    "to": "Public",
    "summary": "Evan's review of 'ActivityPub Is Great'",
    "content": "These people have good taste in protocols."
  }
}
```

You can see that the `object` uses an `article` type with a URL rather than an `id`. This is great for web content, but `id`s are preferable for all objects; you could use a `Link` object here too. The response to the media experience is in the `result` property.

Another media experience activity type is `Listen`, for audio:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Listen",
  "to": "Public",
  "actor": "https://social.example/users/evanp",
  "object": {
    "type": "Audio",
    "url": "https://open.spotify.com/track/3UBItNVbFQiVC5hBQlBvnr",
    "name": "Your Woman",
    "attributedTo": {
      "id": "https://mas.to/@Jyoti",
      "type": "Person",
      "name": "Jyoti Mishra AKA White Town"
    }
  }
}
```

Similarly, you can use the `View` activity type for viewing videos, TV shows, or movies:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "View",
  "to": "Public",
  "actor": "https://social.example/users/evanp",
  "object": {
    "type": "Video",
    "url": "https://www.imdb.com/title/tt4607980/",
    "name": "Orang-U: An Ape Goes To College",
    "attributedTo": {
      "id": "https://social.coop/@mattl",
      "type": "Person",
      "name": "Dr. Matt Lee"
    }
  }
}
```

An issue with media experience is using a controlled vocabulary of IDs for media as well as for media creators. If the content is on the web, you can use the `url` property, but an ID is preferable. You can also fall back to the `Link` type, which makes it clear that the entity does not have an ActivityPub ID.

Another option is to develop and build vocabulary services on big databases for books, movies, songs, and so on that can provide IDs and information about the media in Activity Vocabulary format. As of this writing, there aren't a lot of services like this, but we can hope.

The Activity Vocabulary types point in the right direction, but there's more to the social web. Next, I'll introduce other well-known vocabularies you can use for your ActivityPub software.

Using Other Well-Known Vocabularies

Your chances of interoperating with other ActivityPub implementations shoot way up if you use well-known vocabularies. But finding the right AS2 extension to use isn't always obvious. One way is to inspect the activities coming through your server or client software. Often other fediverse software uses types or properties that you might be interested in using too.

The SocialCG at W3C maintains a collection of extensions, vocabularies, and their uses.

The Fediverse Enhancement Proposal (FEP) process is another effort to create, discuss, and track AS2 extensions. Not all FEPs are vocabulary extensions, but quite a few are, and they are usually well vetted by ActivityPub implementers and fans.

In the rest of this section, I'll discuss a couple of common vocabularies that aren't specific to ActivityPub but can be helpful for expanding the scope of AP objects. They are widely used in the JSON-LD world, so you can easily include them in AS2 documents.

Miscellany

ActivityPub Miscellaneous Terms is a social vocabulary created by the W3C's SocialCG to define some terms that were proposed for ActivityPub but did not make it into the specification in time for publication. It's kind of a grab bag of unrelated terms, united only in the fact that they are widely used on the fediverse and not part of the official AS2 spec. This vocabulary has a context document at *https://purl.archive.org/socialweb/miscellany*, but the terms are actually in the main AS2 namespace.

Most important is `Hashtag`, which we discussed in Chapter 2. Like `Mention`, a hashtag is a subtype of `Link` and has similar properties. Here's an example of a `Note` with a hashtag:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://purl.archive.org/socialweb/miscellany"
  ],
  "id": "https://discussion.example/note/338",
  "attributedTo": "https://discussion.example/user/17",
  "to": "as:Public",
  "type": "Note",
  "content": "<a href='https://tags.example/example'>#example</a>",
  "tag": {
    "type": "Hashtag",
    "href": "https://tags.example/example",
    "name": "#example"
  }
}
```

Hashtags are an important part of searching and organizing information on the social web. They were proposed by Chris Messina in 2007 and have since been used on many social networks to provide a simple, unstructured categorization of content.

The miscellany context has a few other properties. For example, `manuallyApproves Followers` is a property for ActivityPub actors that indicates whether `Follow` activities will be approved by the person directly or the server will do it automatically. Mastodon and other AP servers show this type of account as *private*.

The `movedTo` property is used for data portability to indicate that an actor has moved to another account.

Finally, the `sensitive` flag can be added to an object to indicate that the content might require informed consent on the viewer's part. This is obviously subjective, of course, but in fediverse culture, erotic content, political discussions, graphic violence, and even pictures of food or alcohol are often marked sensitive. When a content object is marked sensitive and it has a `summary` property, some clients will show the summary as a content warning and blur or hide the content.

Here's an example of using a content warning:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://purl.archive.org/socialweb/miscellany"
  ],
  "id": "https://social.example/note/1",
  "type": "Note",
  "attributedTo": "https://social.example/user/2",
```

```
  "to": "as:Public",
  "summary": "Food, meat",
  "content": "I had turkey burgers for supper.",
  "sensitive": true
}
```

The `sensitive` property doesn't follow my recommendations for extension properties; a client that is unaware of the flag's meaning will display the whole `Note` with no warning or blurring. It's good for client applications to support it, though.

vCard

The widely used vCard specification, which defines a file format for personal contacts, is for transferring contact information by email or over other media. In the early 2010s, the Semantic Web Interest Group adapted the names and formats of properties in the vCard format into RDF so that vCard properties and types can be used directly in ActivityPub documents. Dozens of types and properties are described, but I'll point to some of the most useful here in the context of ActivityPub.

The `vcard:` namespace is built into the AS2 context document, so you don't need to include it directly. You can just use the `vcard` prefix on your properties that come from that vocabulary. One use is to enhance information about people:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": ["Person", "vcard:Individual"],
  "name": "Evangelo Stavro Prodromou",
  "vcard:nickname": "Evan",
  "vcard:tel": "555-555-1234"
}
```

Remember, though, that using private personal data for real people on the public social web is generally a bad idea. There aren't many easy ways to preserve someone's privacy when their contact info is used in their actor profile. You can try showing different information when the requester is authenticated and a known contact, but this kind of contact info should never go into an `actor` or `attributedTo` property unless you really intend to share it.

Businesses can enhance `Place` with their contact information:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": ["Place", "vcard:Location"],
  "name": "Barbara's Fishtrap",
  "vcard:street-address": "281 Capistrano Road",
  "vcard:locality": "Half Moon Bay",
  "vcard:region": "California"
}
```

Using business addresses or other contact information is usually much less of a privacy issue than for individuals, especially for retail outlets. Businesses usually *want* to be found!

Other properties define relationships between objects. The `hasMember` property can show members of a group or organization, for example:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": ["Organization", "vcard:Organization"],
  "name": "Social Web Incubator Community Group",
  "vcard:hasMember": "https://cosocial.ca/users/evan"
}
```

In addition, an extensive relationship vocabulary describes relationships between people, which can be useful with the `Relationship` type in AS2:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Relationship",
  "subject": "https://social.example/users/ernie",
  "relationship": "vcard:Coresident",
  "object": "https://social.example/users/bert"
}
```

vCard is a big vocabulary, but it should probably be the first place you look for extra properties about people, groups, and organizations. It is the only vocabulary recommended in the AS2 specification as a way to enhance those types of objects.

Schema.org

Another fascinating vocabulary, provided by Schema.org, covers a wide range of types with a rich set of properties. It was originally developed for marking up web pages with semantic data but has expanded to become one of the default namespaces in the JSON-LD world.

This vocabulary covers a lot of the same kind of types that Activity Streams does—creative works like articles and images, as well as people, organizations, locations, and events. It also includes activity-like types called *actions* and a deep medical vocabulary that's unlikely to apply for ActivityPub uses.

The vocabulary is extensive enough that the question often comes up: since Schema.org has many similar types, why does AS2 need to exist at all? The answer is that their type hierarchies aren't exactly identical; AS2 has social activities not covered in Schema.org. Also, when AS2 was being standardized, concerns arose about the intellectual property rights on Schema.org that have since been resolved. Finally, AS2 is derived from Activity Streams 1.0, which came out several months *before* Schema.org. The two vocabularies have a lot of commonality, but also enough differences that AS2 was really needed.

The `schema:Person` and `schema:Organization` types include useful additional properties:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "schema": "https://schema.org/" }
  ],
  "id": "https://social.example/users/evanp",
  "type": ["Person", "schema:Person"],
  "schema:jobTitle": "Director of Technology"
}
```

More interesting, though, is the deep hierarchy of classes around `schema:Place`. These include administrative areas like cities or states, businesses, and physical features like bodies of water:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "schema": "https://schema.org/" }
  ],
  "id": "https://geo.example/ca/qc/rivers/st-francois",
  "type": ["Place", "schema:RiverBodyOfWater"],
  "name": "St. François River"
}
```

Event types have a similarly deep hierarchy, which can provide additional context to AS2 `Event` objects:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "schema": "https://schema.org/" }
  ],
  "id": "https://schedule.example/party/2024/03/04/11821",
  "type": ["Event", "schema:SocialEvent"],
  "name": "A party"
}
```

One point to note: the preferred version of the Schema.org namespace URL is *https://schema.org*. However, some older documents and software use the *http://schema.org* URL instead, with *http* instead of *https*. For maximum interoperability, publishers should use the newer *https* URL; consumers should be able to handle both.

In general, Schema.org types and properties are a great way to extend AS2. They are among the most-used terms, and a great catalog of them is available. But if you can't find a type that perfectly fits your needs, the next section describes how to create your own vocabulary to use with AS2.

Dublin Core

The *Dublin Core Metadata Initiative* (*DCMI*) defines a set of metadata terms for all kinds of online and offline content. *Metadata* here is data about the data—describing digital objects, their types, contents, authors, and so on.

A lot of the Activity Vocabulary is about metadata: who did what, when, where, and how. Even so, bits and pieces of Dublin Core data may be useful. The vocabulary is widely used, so if you want to express something about the data, it's a good idea to start with the Dublin Core terms.

For example, you may want to include a property of a `Link` that gives the size of the file. We don't have a property for this in AS2, but you can use the `extent` property from Dublin Core to fill in the gap:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "dc": "http://purl.org/dc/terms/" }
  ],
  "type": "Link",
  "mediaType": "image/svg+xml",
  "href": "https://activitypub.rocks/static/images/ActivityPub-logo.svg",
  "dc:extent": "19.1 KB"
}
```

Another useful Dublin Core term is for defining the content license, that is, reuse information about a work. Content licensing is complicated, but for people who want to share their creative works with the world, Creative Commons (CC) has a set of free-to-use content licenses. CC has licenses allowing reuse with attribution, non-commercial reuse, verbatim reuse, and more. Here's an example of using the Dublin Core license property to specify the CC BY-ND license, which allows redistribution of the work verbatim, with attribution:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    { "dc": "http://purl.org/dc/terms/" }
  ],
  "id": "https://social.example/users/evan/note/101",
  "attributedTo": "https://social.example/users/evan",
  "to": "as:Public",
  "type": "Note",
  "href": "Getting my summer tires installed.",
  "dc:license": "https://creativecommons.org/licenses/by-nd/4.0/"
}
```

As I mentioned in Chapter 2, you should stick with AS2 terms whenever possible, and use other vocabularies to fill in gaps and enhance the information. This is especially the case with Dublin Core, which has so many terms that overlap in meaning, or even spelling, with AS2 terms.

Creating a New Vocabulary

Once you've checked all the most likely spots and you're *sure* there's not a vocabulary that meets your needs, you may want to create your own vocabulary. This is straightforward, especially for people used to object-oriented software design.

One antipattern that some developers fall into is defining a namespace for their own project, like *mysoftware.example*, and then putting a whole jumble of unrelated terms in that single namespace. This makes it hard to reuse your designs if you're working on a problem area that others might have. And really, if you're including the information in your ActivityPub objects, that's a clear situation where interoperability is important!

The other antipattern is to just make up terms and insert them into your documents without an extension context. Surprisingly, this will usually work, since the terms are just assumed to be part of the default AS2 namespace. But if you run into naming conflicts with other extensions that have pulled the same trick, you're going to have problems. Namespaces are helpful; that's why we use them.

This section documents the best practices in creating an ActivityPub extension. For this example, I'll consider an extension that lets actors track their running workouts.

Defining the Terms

In our example, the main activity an actor will do is `Run`. We'll define `Run` as an activity type, with its closest related type as `Travel`. We'll use the beginning and endpoints as the `origin` and `target`.

Another important part of running is the route, the line the runner follows. For us, a `Route` will have a `totalDistance` and a set of `waypoints`—that is, the `Place` objects the runner visits. On a typical run, the actor will visit only a few waypoints, so it's fine for this exercise to use an array instead of a paged collection. Rather than reusing `location` or `object`, which would be confusing for processors that understand only the `Travel` type, we'll add a new `route` property.

We can use the standard AS2 properties `startTime`, `endTime`, and/or `duration` to track time. We can also throw in a `splits` property to show the amount of time each segment of the run takes. We'll add a `Split` for each of these items, each with a `cumulativeDistance` and `duration` property.

Finally, we'll add a property for `averageHeartRate`, assuming the runner has a device that can track and report it, such as a smartwatch.

It's common in ActivityPub-related vocabularies to use a simple naming convention for terms. Types usually are capitalized, like `Route`, and properties are usually camelcase (initially lowercase, with subsequent words appended in uppercase), like `averageHeartRate`. No hard-and-fast rule exists, and the exceptions abound, but it's a good rule of thumb to stick to.

Here's an example `Run` activity in this system:

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://namespace.example/ns/run"
  ],
  "id": "https://social.example/users/evanp/run/128",
  "type": ["Run", "Travel"],
  "to": "https://social.example/users/evanp/followers",
  "actor": "https://social.example/users/evanp",
  "origin": "https://places.example/ca/qc/montreal/laurier-metro",
  "target": "https://places.example/ca/qc/montreal/parc-mont-royal/belvedere",
  "startTime": "2024-03-04T12:00:00Z",
  "duration": "PT55M",
  "averageHeartRate": 124,
  "route": {
    "id": "https://social.example/users/evanp/run/128/route",
    "type": ["Route", "Object"],
    "totalDistance": 8.3,
    "waypoints": [
      "https://places.example/ca/qc/montreal/laurier-metro",
      "https://places.example/ca/qc/montreal/monument-george-etienne-cartier",
      "https://places.example/ca/qc/montreal/parc-mont-royal/belvedere"
    ],
    "splits": [
      {
        "id": "https://social.example/users/evanp/run/128/split/7",
        "type": "Split",
        "cumulativeDistance": 7,
        "Duration": "PT423S"
      }
    ]
  }
}
```

Defining the Context Document

The next step is to define a JSON-LD context document for the extension. JSON-LD context documents are straightforward. I recommend looking at examples, and checking the JSON-LD website for details if you get stuck. Here's my stab at a context document for the running vocabulary:

```
{
  "@context": {
    "run": "https://namespace.example/ns/run#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "Run": "run:Run",
    "Route": "run:Route",
    "Split": "run:Split",
    "averageHeartRate": {
      "@id": "run:averageHeartRate",
      "@type": "xsd:nonNegativeInteger"
    },
    "totalDistance": {
      "@id": "run:totalDistance",
      "@type": "xsd:float"
    },
    "cumulativeDistance": {
      "@id": "run:cumulativeDistance",
      "@type": "xsd:nonNegativeInteger"
    },
    "route": {
      "@id": "run:route",
      "@type": "@id"
    },
    "waypoints": {
      "@id": "run:waypoints",
      "@type": "@id",
      "@container": "@list"
    },
    "splits": {
      "@id": "run:waypoints",
      "@type": "@id",
      "@container": "@list"
    }
  }
}
```

Most of this process is just defining abbreviated terms in the context. Each term uses the `run:` namespace prefix. The namespace is defined as the URL for the context document plus a hash (#) separator.

For the types, we just need to define the term and we're done. The properties are harder; each needs to have a type. The scalar properties, like `averageHeartRate`, get their types from standard XML schema types. For properties that have object or array

values, the type is just `@id`. Finally, any properties that have lists need a special `@container` property indicating they're an ordered list.

There's not a ton of constraints in there—no domains and ranges for the properties, for example. We can capture those later during documentation, or if you're feeling adventurous, you can try defining those in OWL, the Web Ontology Language.

The context document needs to go up to a permanent storage URL; *purl.archive.org* and *w3id.org* both provide long-lasting URLs for namespace documents like this.

Publishing the Documentation

The final step is documenting the design and sharing it with other implementers. The FEP process, as I've mentioned, is a great structure for building extensions. Proposals are reviewed by other implementers directly, and their feedback can improve the design. Another possibility is using the W3C's ReSpec format to create your own document.

Any ActivityPub extension documentation you write should include the motivation for the extension; definitions for each term, including range and constraints on properties; and illustrative examples of documents. Don't forget to use *.example* for your IDs and URLs!

You may also want to consider including a license, so other implementers know they're not infringing on your copyright or patent rights by implementing the extension. The W3C Contributor License Agreement is a good template.

You can announce a new extension on the fediverse, or on the SocialCG mailing list. A registry of extensions is on the Activity Streams extensions page on the W3C wiki. It's linked from the AS2 specification, so people looking for AS2 extensions go there first.

Growing an Extension

A context and a documentation page are just an idea. What breathes life into an extension is getting implemented on the social web itself. As a developer, you can obviously start by implementing the extension in your own software. That should prove that the design works and that it provides value for real people. If not, there's no shame in editing and updating your extension until it works the way you want it to. Making backward-compatible changes (say, by adding terms instead of changing them) can make any changes easier.

Protocols become powerful, though, when there are multiple, independent implementers. Spreading the word on your extension might happen naturally; as implementers see your activities or properties go by, they might follow their noses and find the documentation on their own. It's also possible to reach out directly; especially for

open source implementations, adding an issue on the project's GitHub repository that asks for an implementation is a great way to start the conversation.

As adoption grows, the extension could become part of the default AS2 context. The SocialCG has a policy on adding new types and properties to the context document, requiring two independent publishers and two independent consumers. If that criterion is met, and the context, documentation, and licensing all match the needs of the ActivityPub community, the SocialCG will add your extension so that it's like part of AS2 itself.

Conclusion

Extensions are the engine of innovation in the ActivityPub world. Well-behaved ActivityPub servers and clients can gracefully handle extension types or properties that they haven't seen before. This lets other servers and clients include extension types and properties that flow through the entire ActivityPub network. Well-known vocabularies exist beyond the ones defined in the ActivityPub spec—from the Activity Vocabulary, vCard, Schema.org, and others. And as a last resort, you can exercise your own creativity and publish a new extension vocabulary that your software and others can use.

CHAPTER 6
Far Horizons

So far in the book, I've talked about ways of interacting with ActivityPub that already exist—either as defined in the ActivityPub and AS2 specifications, or in the implementations that exist in the world. In this chapter, I want to point a little further afield and sketch out potential uses for ActivityPub in areas so far unexplored or lightly explored.

I hope this chapter inspires you by offering ideas and thoughts that take you in new directions. If you find here a discussion that you'd like to follow up on, or a way to use ActivityPub in an area of technology that you work in, I encourage you to reach out to other practitioners to start sketching out an extension as described in the previous chapter.

The abstract model of the ActivityPub network is robust. Every ActivityPub object is addressable over HTTPS (at least). ActivityPub actors are activity emitters and receivers. They create and modify addressable objects based on the type and properties of the activities they receive. The ActivityPub network uses a push notification model, so that new activities generated in the network are delivered to their addressees in near real time. ActivityPub has a built-in publish/subscribe model (with followers and following), and the most likely future uses will leverage that model, but other modes of interaction can be added too.

Many of the applications I discuss in this chapter have been experimented with or discussed publicly, but most are not widely used or implemented. That's what makes them *future* parts of the fediverse. If you find them intriguing, *please* feel free to take the ball and run with it. Start a protocol discussion, build sample implementations, and consider an ActivityPub extension to cover the behavior you find exciting.

Near Horizons

Before going off into the wild blue yonder, I want to talk about some of the work actively being standardized as I write this book in 2024. It's a time of exciting work in the ActivityPub world, and a lot of people are developing new features and tools. In rapid-fire order, here are some cool projects that are happening right now:

Data portability
: The `Move` activity mentioned in Chapter 5 is being enhanced with additional copy-like functionality for all the actor's content and activities. It's a big step, making it possible to seamlessly move both the social graph and the content from one compliant platform to another.

Discussion forums
: Web-based forums have a deep history, parallel with and supplementing social-network software. This project is standardizing interaction between forums by using ActivityPub to allow users on one forum to read and post to other forums. Called the *threadiverse* by its devotees, this effort is a big step forward for web forums.

End-to-end encryption
: Data in the ActivityPub landscape is encrypted in flight, via HTTPS, as it moves from client to server, and between servers. However, the data is often stored in the clear. This project will allow ActivityPub API clients to encrypt data so that it's unreadable to anyone except its recipients. It's a big advance for privacy on the social web.

And, as mentioned before, the FEP process continues to generate ideas and implementations. A lot is going on in the fediverse today.

Objects as Actors

We've talked about the various actor types and how they can be used—`Person`, `Group`, `Organization`, `Application`, and `Service`. Each can produce activities, follow other actors, and participate fully in interactions on the fediverse.

But ActivityPub *explicitly* does not require that ActivityPub actors have an actor type. (I'm going to say it again: *I'm sorry* we overloaded that term so much in the ActivityPub specification and that I've replicated the problem in this book. *Mea culpa*!) Other types of objects from the Activity Vocabulary can be used as actors on the fediverse.

As I mentioned in Chapter 4, what makes an ActivityPub actor an actor is that it has the required properties of an actor: `inbox`, `outbox`, `following`, `followers`, `liked`. Giving ActivityPub objects these properties, with their proper behaviors, opens the objects up to being active parts of the social web.

Consider some of these examples:

Content types
: Content types like `Article` and `Document` have a lifecycle that we discussed earlier. They can be created, updated, and deleted—as well as liked, replied to, and shared. It might be interesting to follow what's happening with the document rather than following the document's author. The document wouldn't be the `actor` in most activities, but it could share the activities done to it with an `Announce` activity.

Page and profile
: Web pages are a special type of document that deserves particular attention. Changes to web pages are important but maddeningly irregular. It is a tedious exercise to build software to monitor a web page, polling at different intervals to determine whether anything has changed.

  But a web page can have an equivalent `Page` object that can be followed—for example, by using `link rel="self" type="application/activity+json"` in the `head` section of the HTML. If that object re-shares activities made to change itself, remote software or users can re-fetch the page only when it changes, not every 5 minutes.

  A `Profile` page is a special type of page for a single person or organization. It might make sense to do the same type of follow relationship for profile pages, although the difference between the profile page and the person may be a little too philosophical for most software developers to handle.

Places
: I discussed using the geosocial types in Chapter 5, in which `Place` takes a critical role. As with content objects, a place can have the necessary actor properties to be followable. Places could `Announce` the geosocial activities that include them as an `object`, `origin`, or `target`. They could also re-share content objects that use them in the `tags` array.

I think lighting up the digital world with followable representations is an exciting possibility. Both for automated following software, and for humans, connecting with objects that report on their own lifecycle can be revealing.

For implementers, I think using multiple `type` values on the object, with one value being one of the actor types, can make this process a little more palatable for other fediverse software. Especially for automated objects, `Application` is a good candidate here. So, for example, a representation of a place could have `"type": ["Place", "Application"]` in its code.

Search

One problem with a federated network is that content is being produced all over the place, and it's hard for any one person to get a good picture of what's going on. As of this writing, about 25,000 servers are on the fediverse—each with its own set of users, text, images, and other documents. Many ActivityPub servers include some kind of full-text search functionality, but it's usually restricted to the content produced on that server or to content received by users on that server.

Global search means searching across the millions of user accounts on thousands of servers across the fediverse. In the context of the fediverse, one is particularly interested in the ActivityPub object that matches a search term, so that it's possible to reply, share, or like the content (among other reactions!). Traditional web search engines, like Google, are good for returning web pages that contain search terms, but it's hard to filter the results to return only content from the social web.

Global search has several main areas of interest:

Content search
: If you're interested in a particular topic, like scuba diving or Albania, you can search for text like "scuba diving" or "Albania." Many content search systems can identify related strings like "AL" for the country code, or "scuba divers" for a related term. Global content search would typically show only public content.

Tag search
: Hashtags are a particularly interesting subset of content search. As mentioned in Chapter 2, hashtags are identified by ActivityPub API clients in submitted content and are stored separately in the `tags` property of a content object. Unlike free-text search, tag search is a collaboration with users who are explicitly building an affinity between posts using the same hashtag.

Profile search
: With tens of thousands of servers on the fediverse, finding a particular person you're interested in can be challenging. Especially when getting started with your first fediverse account, finding people you know from other contexts—family, friends, colleagues, neighbors—can be frustratingly slow. People search can help find people you know or people you're interested in, across the entire network. Often people search will include social connections between the searcher and the people in the search results. This helps you judge if the "Jane Jones" in the search results is the same Jane Jones you remember from the lawn bowling club, because some of your other friends from the club also have links to this Jane. (This feature works even if you aren't into lawn bowling.)

Besides differences in search result type, there are differences in the way social search engines work. A *retrospective search engine* looks into the past: it uses search agents that roam about the social web, identifying content that has previously been published, and indexing it for deep analysis. A *prospective search engine* looks into the future: it returns new content or people matching a given search term as that content is being published (sometimes within seconds). Retrospective search is great for research and discovery; prospective search is great for remaining part of the conversation. Both are important for content and tag search. (Thanks to search veteran Bob Wyman for the excellent conceptual framing!)

Real-time search is particularly interesting for ActivityPub in that it could reasonably be implemented in-band as part of the protocol. Representing a search term as an actor, as discussed in "Objects as Actors" on page 212, makes that term followable on the network. An implementation could repost public content it identifies as matching the search term.

One last important distinction between search systems on the fediverse is between centralized or distributed search. In a *centralized search* system, a single service controlled by a corporation or person collects data into its own data store and provides search services to users and software. Google, DuckDuckGo, and Bing are notable centralized search systems on the World Wide Web.

In a *distributed search* system, by comparison, the data store of indexed content or user accounts is spread across many servers or services, with different owners. Typically, a search client requests matches for a search result to a single service, and that service knows how to ask other servers if they have a match for a search term. The networked search services could be the ActivityPub servers where the content was first posted, or they could be dedicated services. Many fediverse fans prefer the idea of distributed search, since they feel it better reflects the decentralized nature of the social web.

As of this writing, search engines do not yet play an important role on the fediverse, despite many users' requests for this type of functionality. For other fediverse users, search engines represent an invasion of their privacy—taking their content, even if it's marked public, and republishing it out of context. In particular, for vulnerable or marginalized communities, easy content or profile search for terms related to those communities exposes members to possible harassment.

You can use these options to identify users who don't want their profiles or content to be indexed: using the Robots Exclusion Protocol, also known as *robots.txt*; the `indexable` extension term for actors used by Mastodon and other ActivityPub servers; and hashtags like `#nobots` or `#noindex` that users add to their profiles. Finding a good balance between searchability and privacy is going to be an important part of the next phase of the fediverse's development.

Artificial Intelligence

Artificial intelligence (*AI*) is a hard technology to define. Here's one definition I like: AI is a set of technologies used when a problem domain has become so complex that programmers can tell the software only *what* to do but not *how* to do it. AI technologies such as expert systems, fifth-generation programming languages, or ML engines are built from robust general *algorithms* that can figure out how to achieve a result, usually, given a set of goals and constraints.

AI has applications for the social web:

Algorithmic feeds
: Many social networks don't show the home feed or `inbox` in reverse-chronological order, as specified in ActivityPub. Instead, the activities there are sorted according to an ML algorithm to achieve a particular result. For many networking services, the primary desired result is to get you to click ads on their site, with secondary results like getting you to stay on the app longer (so you have a higher chance of clicking an ad later) or getting you to engage with others so *they* stay on the app longer (and hopefully, y'know, click some ads).

  Fediverse denizens have developed a healthy distrust of algorithmic feeds. Many that run their own platforms read their home feed only in reverse-chronological order, from newest activity to oldest activity, without any filtering. They report less addictive behavior and more healthy engagement with their community.

  But things don't have to be that way. Algorithms can be put under users' or communities' control and given different goals that serve their needs instead. What about an algorithm that emphasizes overall user happiness by focusing on feelings of meaningful engagement in topics that matter, and strong relationships with people we care about? Or algorithms that have the goal of productive, collaborative community interactions? Using a federated social network means having the choice of which platform and which community to engage in, and it also means having the choice of how best to engage.

  I think it will be interesting to see how algorithms work in fediverse feeds going forward. I'm really interested in having better friendships, and using software tools to help me emphasize them seems like a great practice.

Content moderation
: Moderation occurs when social-network operators remove or deemphasize content that is disturbing or irritating to users. This can be spam, harassment, or other images, text, or videos the operator deems inappropriate for their platform. Moderators can scan all incoming content or can wait until users report content for review, perhaps using a `Flag` activity.

At low volumes, content moderation can be done by hand by humans who carefully consider each piece of reported content to see whether it oversteps platform rules, and choose the appropriate remedy if needed. But as the volume of content on a platform increases, human moderators might have difficulty keeping up with the pace, reviewing hundreds or thousands of reported content items per day.

One solution is to use AI to do a first-level scanning of content. The AI can discard the most obvious and egregious problematic items, and save the more subtle or novel ones for human judgment. This lets humans spend their full attention on a tiny sliver of the most challenging questions, and AI can just dump the rest. And the AI can learn from the human decisions in those edge cases, and handle them automatically if they come up again in the future.

Structuring AI content moderation systems is a difficult social process. The AI will work better if it learns from many humans, but the more people who participate in an AI filtering system, the more chance that opinions will vary about acceptable content. And some organizations and people feel uncomfortable submitting their social content to a remote service for review.

I think that content-moderation AI will probably be installed at various levels in the social web—one AI agent per person, one per network, and perhaps larger ones that service multiple networks. The way they interact and the data they share will be an interesting compromise to see develop.

Generative art

This type of art comprises any images, text, video, or audio that's made by computers without direct human input. Often generative visual art will use an initial state and repeated equations to create dynamic, unpredictable images or video. Generative text is sometimes created by adding randomly chosen words into a text template. Many, but not all, *bots* on social networks produce generative images and text to share with their followers. One of the best parts of the fediverse, as of this writing, is how much the bot-author community has adopted fediverse platforms like Mastodon.

One hard aspect of randomly generated art is that it can be hit-or-miss. Bots usually work on volume; producing 10 or 15 images a day, they get one or two that are interesting or funny to their audience. Agents with more sophisticated algorithms, like large-language models (LLMs), can produce more lively, interesting images and more readable text.

AI bots can take things a step further—learning from feedback like replies and likes, and changing the parameters of their algorithms accordingly. I'm interested to see how bot developers work with AI and increase the sophistication of their projects by including AI techniques into the generation of the work.

Conversational interfaces

As AI becomes more responsive, it can move from the realm of the generative to the *conversational*. That's when the information flows two ways: from a human on the network to the bot and back again. No matter who starts the dialogue, a conversational interface can respond to questions, correct errors, or crack jokes with their human interlocutor.

More importantly, conversational interfaces can connect with other digital systems—both sensors (measuring things) or actuators (doing things). One bot might connect a weather network and could discuss what you should wear to work and whether you should bike or take the train. Another bot may be able to buy you tickets for a baseball game (see "Payments" on page 226) and help you choose the best seat.

Conversational messaging, on the public social web, connects service providers and their users in a natural, humanistic way. It can integrate simply as a messaging transport or perhaps use the broadcast technique of following and followers. I think the combination of widely available LLM networks and active bot development on the fediverse makes this area of development appealing.

Content Management

A *content management system* (*CMS*) is a service that lets one or more people author, edit, and manage sometimes quite large collections of text, images, video, audio, and other content. Some CMSs allow building sophisticated interactive experiences like polls, forms, or maps, and many are extensible with plug-ins that allow even more complex content types.

The intersection of content management functionality with ActivityPub is considerable. The ActivityPub API allows basic content management activities like `Create`, `Update`, and `Delete`. It can support numerous common web content types natively. More important, it has an extensibility mechanism that allows defining new data types and activities for a service.

With such a close mapping, a clear opportunity arises to consider ActivityPub's API as a potential standard for CMS client applications. A standard API would let client software developers implement solutions that work across multiple backend CMS services. An existing open standard for CMS APIs, called Content Management Interoperability Services (CMIS), has broad support in the industry, although it lacks some features for modern applications.

Some mismatches exist between the ActivityPub API and many CMSs. In particular, the author of content in the ActivityPub authorization model reigns supreme and has sole ownership of the content. Loosening that model to allow shared editorial control

of content (for example, by role or group membership) is definitely possible but might require some adjustment or extensions.

Some publishing systems also have a production lifecycle for content—from first drafts, through editorial review, revisions, and then publication and ongoing support. Incorporating these stages in the content lifecycle would require additional modeling—either in terms of new activity types, addressees, collections, or other data structures.

But the usefulness of remote following content can't be overstated in this situation. If authors or even content objects can be followed from other networks, many of the notification tools necessary for content management become more standardized and friendly. For internal teams at the publishing organization, dynamic content tracking can make following the progress of published articles or content much easier and more automatable.

In addition, for public-facing installations that depend on social media circulation to drive demand, being able to integrate content directly into the fediverse would be a killer app. Loosening dependence on intermediary social networks, and taking a first-class place in the social network instead, could be a revolutionary step for publishers.

I believe that CMS software developers have a fascinating opportunity to embrace ActivityPub, both for federation and for a standard API. Some software, like WordPress, has already taken up this chance. It will be interesting to see which others follow.

Games

Many video games benefit from social interactions. A common pattern in builder games, for example, is that a user builds a farm, city, or castle in a virtual world and then connects their creation with those of real-world friends via a social-network relationship. For example, one of the first apps I built using pump.io, a predecessor to ActivityPub, was a farming game where you could plant, tend, and harvest crops on a virtual farm.

With extensible types in ActivityPub, a gamer can create dragons, barns, or sewer treatment plants and share their progress with friends and family. Letting users connect their ActivityPub identity to the game system, and publishing game content out to the fediverse, is a potentially powerful way to increase the popularity and reach of a video-game platform.

Other games may even be playable across the network itself. Word games, board games, or drawing games would be amenable to structuring as ActivityPub extensions. The ActivityPub Primer page on extensions uses chess as an example vocabulary to illustrate how extensions work. Designing gaming protocols that support play

across heterogeneous platforms is a lot more complicated, though, than running on a single game program or platform. Your software can't enforce the rules; you need to design the protocol to make cheating difficult.

Some video-game platforms, like Xbox, Nintendo, and PlayStation, let gamers establish an online identity, create social connections, and track a stream of activities from those other players. In a special part of the platform interface and in pop-up notifications, gamers find out when their friends are starting and stopping games, reaching in-game achievements, or even sending notes, screenshots, or challenges. They can look at friends' profile pages, see badges and scores, and look at their social graph as well. They can invite friends to play.

You don't have to squint too hard to see that these meet the definition of social networks as I laid them out in Chapter 1. Which raises the question: could they integrate with the fediverse, and would it be worth it? I think it would be amazing, giving the players extra reach and visibility, and helping to grow game communities. Is it a bridge too far to hope for inter-platform connectivity, seeing your friend's Animal Crossing updates on your Xbox screen? I don't think so.

Health Tracking

Human beings are singularly good at fooling ourselves. We consistently underestimate our indulgent habits and overestimate our healthy ones. The effect of our built-in psychological weaknesses on our physical and mental well-being is stark. So many of us have the perfect circumstances for a healthy life, surrounded by everything we need, except the ability to correctly choose our best path forward.

One hope I (and maybe you) have is that our electronic devices can help us be more careful and mindful with the bounty that surrounds us. Computers, mobile phones, and other more dedicated devices have so much to offer to help with healthy habits: unrelenting clocks, reliable memories, access to huge databases of exercise and food data, specialized sensors for measurement and monitoring, cameras, and connections to our friends and family for suggestions and inspiration.

Apple, Google, Strava, and other companies provide health-tracking apps for mobile devices. They include ways to track sleep, exercise, eating habits, meditation, and other healthy practices. They also let third-party apps connect through the device API, including dedicated devices like heart-rate monitors or bathroom scales. Some let you share activity with friends and family.

What does health tracking have to do with ActivityPub, though? You don't have to strain hard to see the parallels with social-network applications. Health trackers often show a stream of activities in reverse-chronological order, much like a social-network activity stream. The way that third-party apps and dedicated devices feed new event data into the system is similar to the way social-network client apps use the ActivityPub to feed new

activity data into the server. And, of course, connecting with friends and family and sharing our activities with them is what the fediverse is all about.

The benefits of using ActivityPub for this application are many. First, having a standard API across platforms for a health-monitoring application to generate new activities means much less custom code for those clients and much more choice of applications for users. Picking a health-tracking service based on your own privacy, cool new features, or other needs is another great option for customers. A standard API would let users expose their activity data to third-party analysis, which may give better insights on habits, triggers, and technique improvements. And sharing with people who don't use the same mobile platform or fitness-tracking device as you means better social support for your health journey.

A lot of work remains in this area for ActivityPub to become a viable protocol for connecting health-tracking apps. First, we need extended profile properties for physical characteristics like height, weight, body measurements, age, and so on, as well as privacy controls for these parts of your profile that may differ from those of your name and avatar photo. (ActivityPub doesn't require all requests for an actor endpoint to return exactly the same properties, but services need to implement this feature.)

To move into health tracking, we'll need good vocabulary extensions for measurements—like weighing yourself, measuring your heart rate, or doing a fitness test. We'll also need extended activity types for exercise (like running or playing tennis), sleep, and mindfulness practices (like meditation). These activities have spatial, temporal, and other extents that will need special properties to support—such as a map for a biking route, or the weights curled in each set of a weightlifting session.

Tracking hydration is *probably* not too complex, as long as you're willing consider a few dozen beverages, but tracking eating habits will be harder. Representing the dizzying array of the kinds of food that people eat, each with calories, macronutrients, and vitamin counts, makes for a complicated diet vocabulary. An extensible vocabulary that can support user-submitted food types would probably be needed too.

Privacy controls will need to be high-quality in this case. Finer-grained sharing than just to all followers or the public is necessary, preferably to a small group of friendly, encouraging peers, or even medical personnel like doctors or physical trainers. Mixing health activities into the normal social stream, with coarser-grained privacy control, could leak personal information that puts the user's privacy at risk.

An effort in health tracking on the fediverse may be able to leverage existing vocabularies, but getting this process started would still take a lot of effort. The benefit of greater choice and flexible interoperability may be worth the work. As with most areas of systems development, focusing on a specific area and expanding from there is probably the best approach.

Internet of Things

Similar to the health-tracking domain, the *Internet of Things* (*IoT*) could benefit from a standards-based stream subscription protocol like ActivityPub. IoT is the blanket term used for internet-enabled sensors (devices that detect) and actuators (devices that act) placed around our homes, offices, and public spaces. People using "smart home" technology can get video streams from home security cameras on their mobile phone, or turn off and on their lights or heat, or monitor the status of home appliances.

Most IoT devices connect via a proprietary protocol to internet services hosted by the device vendor. Some will connect to home hub devices with open standards like MQTT (originally *Message Queue Telemetry Transport*), which makes connecting devices from different vendors more reasonable.

But what about an extensible web-based protocol like ActivityPub that extends beyond the walls of the home to the entire internet? People who've followed their own refrigerator or dog door on X or other social networks know how interesting a device's life can be and how well the messaging format can fit into a social-network framework. Using ActivityPub would make this kind of update richer than just plain text and accessible from more than one platform.

As I've mentioned, each object can become a followable ActivityPub actor, generating activities about its sensor settings and sharing them with the appropriate followers. Human actors can receive updates from sensors or can even send AS2-formatted instructions to actuators. IoT devices could be followers too—waiting until the homeowner's bike is minutes away before turning on the front porch lights and playing relaxing music in the living room. Connecting sensors and actuators from different vendors by using open standard activity types would allow new kinds of interactions that closed networks do not.

As with health networks, the essential shape of the infrastructure is already there. What's needed are the activity types that reflect IoT events: lights going on, temperature dropping below a threshold, blenders starting, trackers venturing into well-marked geographical areas. Home hubs could bridge their private networks with the world, or devices could connect directly. This may be another case where starting with a small set of initial activity types and letting new entrants add additional extension types as they're needed can help move the integration forward without getting overwhelmed with the enormity of the task.

Also as with health networks, privacy requirements are high here. Nobody wants a stranger half a world away controlling their lawn sprinklers or checking the contents of the pantry. Although IoT vendors have not always been great at protecting privacy, using standards for input and output at least gives the opportunity for third-party devices to detect and even correct vulnerabilities.

I think this is a potentially cool opportunity for ActivityPub and an interesting metaphor for interacting with the built world. After all, why can't your blender be your friend?

Dating

Dating apps and websites are a popular way for people to find romance. Most dating apps allow a user to set up a profile, in which they describe themselves, their interests, and what they're looking for in a match (or *multiple* matches—polyamory dating apps are an important segment of the market). People add photos or videos; to provide some insight into the dater's life, some apps include an activity stream with thoughts, photos, and videos taken throughout the day. When two people express interest in each other, they're able to start a private chat conversation, which can lead to online or offline romance. If the conversation doesn't go anywhere, either party can end it, permanently.

According to a 2023 poll by Pew Research, 30% of all Americans have used a dating app, and 10% of all Americans in partnered relationships met through an app. Worldwide, about 300 million people use dating apps each year.

But many dating app users complain that they can't find someone who meets their qualifications on their chosen app. That makes sense; after all, there are hundreds of apps, so the pool of desirable candidates in any one app can be limited. But what if people could connect *across* apps?

Federated dating means letting app users view profiles, propose matches, and chat across app boundaries. ActivityPub provides an attractive platform for federated dating. Profiles, connection requests, and private chat are all part of the ActivityPub standard model, so it already has a good start on implementing a federated dating network.

To get all the way there, profiles will need more detailed properties that express aspects of the person's physique or personality that dating app users want to see: height, hair color, gender, age, religion, political affiliation, smoking and drinking habits. And federated search services will need to find users matching the searcher's criteria across different apps and services.

Dating requires progressive revelation of private data—something that ActivityPub allows but that isn't widely implemented. Revealing only the basic profile properties for public searches, with additional profile properties shown when a connection has been made, gives users a level of control over their personal information.

One interesting aspect is having a standard API to access the dating platform. This opens up the dating data to third-party apps that can provide advantageous user experiences outside the default product.

By letting daters connect across app boundaries, dating apps could increase the value of their platforms to their users and compete on features. New dating apps can start and have instant access to the full audience of existing potential matches, lowering the barrier to entry for innovative technology.

I think dating apps are an important growth area for the fediverse. Those we love and care for are a serious, essential part of our lives. Creating safe and effective interfaces will take a lot of work and user feedback, but I believe a big payoff can result for the whole industry and its users.

Enterprise Software

Enterprise applications may seem like the last place you could apply social-network standards. The enterprise is for Serious Business, and social networks are for fun and games at home.

But enterprise social networks (ESNs) remain extremely popular in the enterprise. With similar features to—or occasional integration into—enterprise chat, ESNs let team members collaborate in the ways they're used to from consumer social networks. Dozens of software vendors provide ESNs to their customers.

Some of the same problems arise inside the enterprise as we've seen on the public web, however. Connecting ESNs, either within an organization or across organizational boundaries, can be difficult with the same software stack, and impossible across platforms. Similarly, third-party software developers have to make a big bet on a particular ESN platform, as their APIs are wildly different.

ActivityPub can help on both counts, and I expect that enterprise deployments will be an important part of the fediverse as it grows. Using the ActivityPub protocol to follow and interact with partners at other companies will make for more direct connections with finer-grained control by users and administrators.

ActivityPub may also blur the lines between enterprise and public social networks. Many people working in the enterprise also maintain a social presence on LinkedIn, X, or other public social networks. Federated ESNs lets those staff members run their social presence from within the ESN, with customers, partners, and vendors following remotely. This will be a cultural challenge for some organizations that have gotten used to thinking of the ESN and public networks as two different worlds. But just as team members have learned cultural practices around email, they can learn the best ways to remain professional on the ESN.

Another interesting internal and remote interaction is exposing parts of the production process as ActivityPub actors and activities. Most people familiar with online commerce have had the experience of following an order as it moves through the provisioning, manufacturing, and delivery phases. Giving update notifications inside

the company and out can be a great use case for ActivityPub. And having a standard API means that the developers making the manufacturing or shipping software can integrate with multiple platforms with one codebase.

Giving people and equipment in the enterprise an identity on the social web, and letting those actors produce activities to be received internally and externally, may be the killer app for ActivityPub.

Software Development

Software development is a social activity. Programmers use communication tools like documentation, email, and chat to share ideas. They use bug-tracking systems to determine what needs to be done and by whom. And they use tools like Git and pull-request managers to organize and implement their collaboration on source code.

Some software development platforms, or *forges*, like GitLab or GitHub provide activity stream APIs for users, groups, and projects. Many include an identity and a profile page for each participant, as well as a way to follow the activities of other participants.

These social software development features map fairly directly onto concepts I've used in this book for ActivityPub. Software development platforms would probably benefit from implementing the ActivityPub API, so that third-party developers could write their plug-ins or apps once and then use them on different services.

But *federated software development* also has a clear benefit. It would let users on one forge contribute to software projects being developed on other forges. Besides their flagship public services, many software forges often allow private or enterprise deployments. So even for a single vendor, implementing a federation protocol could be helpful. But implementing federation across platforms could definitely unlock even more creative interactions, both for commercial software development and for open source.

As a side benefit, using ActivityPub for building a federated software development network means that forge activities can be followed and integrated into other feeds on both public and enterprise social networks. Sharing notifications of bug reports or new software versions can keep the entire developer and user ecosystem informed.

Many of the types used in software development (like a person, a profile page, or a comment on an issue or a pull request) have easy equivalents with the Activity Vocabulary. Others, like source code, would need to have extensions developed for them. Modeling complex interactions like a pull request review might require significant work. The authorization model is also somewhat different from the base model, in that many people can update or even delete the same object. And the whole area would need to be harmonized with the Git model of distributed software development.

Federation of software development is a challenge, but I think it's worthwhile. As of this writing, at least two platforms, GitLab and Forgejo, are working on integrating ActivityPub. I hope to see progress on this work to allow integration with other social software and real collaboration across forges.

Payments

Humans like money. Some digital platforms include ways for participants to pay each other. This can be for buying and selling digital or physical goods or services (see "Marketplace" on page 228), or for digital subscriptions, tips, or donations.

Often platforms take a percentage out of any money transferred—anywhere from a few percent up to a third or more of the transaction, depending on the platform and the transaction's purpose. Federation could make payments more affordable. Users on the fediverse, including payers and payees, have a choice of which platform to use. That level of choice could make transaction fees cheaper as platforms compete for users.

For payment integration into the fediverse, I see a few options, not necessarily mutually exclusive:

Integrate existing payment platforms
: Building in support for existing payment platforms could be a great first step to having payment systems. A fediverse user could add their PayPal account information to their ActivityPub actor account, as a link or as an extension property, and any activity that initiated payment could be executed out-of-band through the PayPal network. Similar accounts exist for services like Venmo, Cash App, and Interac.

  One problem with this system is that a lot of payment services exist, and most aren't compatible. A user with a Cash App account can't send money to a user with an Interac account. Lower-level payment systems, like the Visa network, are harder to integrate but might be the only way to make payments available across networks.

Integrate blockchain payments
: Another option is integrating blockchain payments. Blockchain currencies, like Bitcoin or Ethereum, are a decentralized way to send and receive payments through a peer-to-peer network. Each user of a blockchain currency can have one or more *wallet addresses* that others can use to send them money.

  Integration with blockchain currencies can be quick. As with conventional payment platforms, the user could add their wallet address to their actor profile as an extension property, and remote users could find and send money to their wallet.

Remote users' software might use the transaction ID as an extension property sent along with activities to prove their payment.

The problems are many, though. First, blockchain payments can take a lot of time to clear, because of the nature of the network, although systems such as Lightning Network can reduce the wait time. Second, blockchain payments can have high transaction fees, based on how willing network nodes are to process and record the transaction. Again, this has been mitigated in some systems.

Blockchain currencies are also volatile. As of this writing, the price of Bitcoin in US dollars has soared more than 500% in the last five years, accompanied by a few crashes that cut the value by half or two-thirds. Volatile currencies make it hard for people to judge reasonable prices or to feel safe keeping their money in a wallet for any period of time.

A lot of friction arises in transferring fiat currency like US dollars into a blockchain currency and back out again. Users have to buy their blockchain currency at an exchange and submit their government IDs to know-your-customer (KYC) reviews. Someone making a purchase on an online marketplace might not want to jump through these hoops just to buy a secondhand pair of maracas.

Most of all, a *lot* of blockchain currencies are out there—hundreds or even thousands more than conventional payment platforms. You can't directly transfer funds from one chain to a wallet on another chain. Most ActivityPub users would need to coalesce around a single blockchain payment technology, like Lightning, to make this work.

Make up something new

Another option is to make up something new, specifically for ActivityPub. Neither traditional banking nor peer-to-peer blockchain payment systems match the federated structure of the social web. An ActivityPub-specific mechanism could fit better within the fediverse framework.

As an example, ActivityPub actors could have a balance of some kind of points to use for exchange on the network. Point transfers between actors could happen in-band; others could review the transaction history to make sure the account balances were correct. Humans would need a way to buy these points and a way to convert them back to dollars (or yen).

A few problems arise here. First, doing this right would be hard, and doing it wrong would cost people a lot of money. Kickstarting the exchange infrastructure would be a hassle and would probably eventually require all the overhead of a blockchain-based exchange anyway.

I think the most likely scenario will be a mix of commercial payment systems and a few blockchain ones. Although this may slow the growth of payments on the fediverse, it may be an unavoidable result of cautious entry into this arena.

Marketplace

A *marketplace* is a technology that lets buyers and sellers browse, negotiate, and close deals on physical and digital objects. Marketplace platforms like Craigslist, eBay, Amazon, and Facebook Marketplace let sellers make *listings* for products or services they provide, and entertain bids from buyers who want to spend money or barter for them. Platforms charge a fee to buyers, sellers, or both to participate in the marketplace.

Social marketplace platforms have additional benefits. First, the marketplace experience is embedded into a social-network UI: you can follow listings, sellers, or buyers the same way as you would follow traffic updates or joke bots. Second, the social graph provides additional social proof for the reputation of a seller or buyer. If you know someone in common with the transaction partner, or someone you know has transacted with this partner before, it can engender the trust necessary for completing a sale.

A *federated marketplace* connects buyers and sellers with accounts and listings on different platforms. It broadens the pool of available products and grows markets for niche sellers. Finding a used bumper for a 1972 Citroën DS on one market might be impossible, but spreading the search across multiple markets makes it more feasible.

Federating marketplaces opens up the choice of platforms, and choice brings competition. Platform competition that doesn't depend on lock-in of buyers and sellers means better platform fees and more useful features.

Payments are an enabling technology that make social marketplaces easier. They're not absolutely required (many marketplaces leave payment handling up to the participants), but they do help for a smoother transaction experience.

Some parts of a marketplace schema are already in place for ActivityPub. We have the `Offer`, `Accept`, and `Reject` activities, plus discussion features like comments and direct messaging. But a lot remains missing: asking prices, reserve prices, closing a deal, making a payment, and ratings for products, buyers, and sellers. We'll also need full vocabulary for all the kinds of physical objects to buy or sell, from baseball cards to real estate. I think this is a fruitful area for exploration in the ActivityPub world.

Happiness

This may seem a strange way to end a section on the future of the fediverse, but it's also the one that's most important to me. Social networking was created by people who wanted to do good in the world—connect existing friends and make new ones, increase understanding, enable collaboration. We believed we were building a techno-utopia. But for many people today, social networks are a source of anxiety, depression, and anger.

If we are rebooting the social-network system, it might be time for us to build in tools to make ourselves less miserable—maybe even, dare I say it, *happy*. Federating networks gives us a chance to make choices about our platforms, such as the features they provide and the algorithms they use. We can even group together and build our own platforms. Why not choose and build platforms that make us happy?

It's not immediately clear what makes people happy in social software and what doesn't. But the psychology of happiness has been progressing by leaps and bounds over the last few decades. The founder of the branch known as *positive psychology*, Murray Seligman, proposes five factors that make our lives fulfilling, called the *PERMA model*:

Positive emotion
: This seems pretty obvious, but feeling pleasure, joy, laughter, pride, and love makes us happier. Implementing features that bring us closer to these feelings, like algorithmic feeds optimized for positive emotion, might help increase this part of our happiness.

Engagement
: This is the sense of being fully engaged in a task. Many social-network platforms bounce us around from idea to idea, topic to topic. They keep news and jokes and updates coming at us fast, like moving obstacles in a video game. Having the time to settle into a task, make something interesting or artistic, and getting to the heightened sense of awareness called *flow*, would make social-network experiences better for our happiness. Cutting back on the distractions in interfaces, improving the tools for creation, and encouraging deeper interaction are all ways to increase this kind of engagement.

Relationships
: Social networks are all about relationships: we add friends, follow our heroes, groom our social graph. So why does it seem we always find ourselves arguing with strangers about trivial topics that don't matter to either of us?

  Social-network platforms could do a much better job helping us concentrate on the relationships that really matter—our household, our extended families, our friends, colleagues, neighbors. Federation means we can connect to everyone

who matters to us, not just those who actively use the platform we're currently on. Putting these relationships front and center, and including features to deepen and strengthen those relationships, could help us have the kind of social safety net that leads to happier lives.

Meaning

One downside of social media is a sense of the triviality. Many of us have walked away from a long social media session feeling like we've wasted that time on topics that didn't matter, that we've somehow become worse people because of the engagement we had.

But all of us have careers, hobbies, beliefs, and goals. We have meaning in our lives. Connecting our social platform experience to those meaningful goals could make us feel that our social participation has more purpose. It might be as simple as following topics related to one's career, or having recommendations for ways to practice crafts or make objects, or being able to participate fully in clubs or political parties we care about. Deep, genuine relationships bring meaning too.

Social media isn't something we do to escape real life anymore. Real life has fully infused itself into social media. We need to be using those tools to embrace and not escape the lives we lead.

Accomplishment

Accomplishing something feels good. Measurable achievements are an extremely powerful way for us to feel like our lives are going the right way.

But social media seems to wipe the slate clean each morning, as a new wave of discourse floods into our feeds. Nothing we've done in the past seems to matter in the real-time world. The best we can do is try to keep track of the metrics available from the platform: number of followers, number of likes and shares. These might not be the metrics that matter to us, but they're the ones we have.

I think we can do better, giving people a sense of accomplishment in social networking by having more permanent achievements linked from profiles and other parts of the network. Badges, awards, testimonials, and other forms of recognition have been part of social networking in previous eras but have mostly fallen by the wayside.

Delving deeper into one's own social-network history can also give a sense of achievement. When did you solve someone's seemingly impossible problem? When were you kind to someone who needed it? When did you bring attention to an issue and rally group action in a meaningful way?

Psychologists also emphasize other aspects to happiness: physical well-being, with good diet, sleep and exercise, is a common addition.

Overall, I hope that we consider happiness and human flourishing in our designs for next-generation social networks. We don't have to replicate the structures and cultures of existing social networks; we can set our own path.

Conclusion

My most sincere hope is that the tools I've discussed in the preceding chapters, and the ideas of this one, have excited a creative drive in you. You stand at the threshold of a new world in social software, and as a developer you have a chance to make a real difference.

One great part of a social web is that you don't have to build the whole thing from scratch; just building a great drawing app and connecting it to the network as an ActivityPub client makes the entire network better, without having to do the other 99% of building a social network.

Thanks so much for taking the first steps on this journey. I can't wait to see what amazing projects you build.

APPENDIX A
Activity Vocabulary Types

Each object type in this appendix has the following information:

Information	Explanation
Extends	Any other type it extends (if any).
Group	Rough grouping, not necessarily reflected by the type hierarchy.
Importance	The relevance of this type for most developers: high, medium, or low.
Notes	Notes on what the type is used for.
Properties	The most relevant properties for this type.
To publish	Notes for publishers to maximize interoperability.
To consume	Notes for consumers to get the best behavior.
See also	Related types from the Activity Vocabulary and equivalent types from other vocabularies, if any. These might be useful places to search for properties that might be missing with this type.

A lot of this information is my subjective opinion. I have a few biases: I want to see maximum interoperability between ActivityPub and AS2 implementers, and I want to see more implementations of cool applications on top of ActivityPub. If those aren't your priorities, my opinions might not make sense to you. *Caveat emptor!*

I'm also one of the editors of the Activity Vocabulary spec, which is considerably more terse and general about most of these types. Whether we should have more specific, more elaborate descriptions in the AS2 specifications is an open question, but you should get a pretty detailed description here.

As you may remember from Chapter 2, Postel's law for interoperability, by the internet's great standardizer, Jon Postel, states, "Be conservative in what you send; be liberal in what you accept." I follow this principle in the notes for publishers and consumers. You will often see in this appendix recommendations for publishers to avoid a particular property or structure, followed by a recommendation for

consumers to be ready to see those same properties and structures in their input. On an open and distributed network with heterogeneous clients and servers, taking a little more care to enable communications can help more people connect to the people they care about.

Groups

I assign each object type one or more groups. Some of these groups are related to parts of the Activity Streams and ActivityPub specs; others are more subjective. Either way, I hope they help you think about the various types and whether you need them for your implementation.

Group	Type
Abstract	Types that are unlikely to be used on their own but provide scaffolding for other types
Activity	Types representing an activity
ActivityPub	Types defined for use with ActivityPub
Actor	Types that are usually used for actors
Collection	Types related to collections and collection pages
Collection Management	Activity types for managing collections
Content Experience	Types for recording the experience of content like audio or video
Content Object	Digital content like text, images, audio, and video
Core	Fundamental types in ActivityPub
CRUD Activity	Types for managing object lifecycle: `Create`, `Update`, `Delete`
Event Management	Types for organizing real-world events
Geosocial	Types for recording and sharing people's travels through the world
Group	Types for an online group of actors that have discussions and share content
Intransitive	Types without an object
Link	Types defining links on the web
Meta-activities	Types for activities that manage activities
Moderation Activity	Types for moderating people's behavior on the social web
Object	Neither actors, nor collections, nor activities, as defined in the AS2 spec
Question	Types to support polls and open-ended questions
Reaction Activity	Ways that actors react to content or activities
Real World	Representing objects or events in the real world
Relationship	Activities and object types for defining relationships between people and other actors
Social Graph Activity	Managing the follower/following graph

Accept

Extends	`Activity`
Group	Activity, ActivityPub, Meta-activities
Importance	High
Notes	This activity type signifies that the actor has accepted the object. Typically, this is used when the actor's consent is requested and the requester needs to be notified. The most important use of `Accept` in ActivityPub is with a `Follow` activity as the `object` property—meaning that the actor has accepted the follower and will send activities to them as a new subscriber. Other likely `object` types are `Invite` and `Offer`. If the actor is accepting an object *into* something, you'll use the `target` property. For example, if Alice accepts John into a `Group`, John would be the `object` and the group would be the `target`. Another example is accepting a submission to a collection, such as to `Accept` a `Note` into the `replies` collection of an image.
Properties	`actor`, `object`, `target`
To publish	• It's not necessary to `Accept` every single activity received. An `Accept` activity is appropriate only if the recipient's consent is necessary for further processing. • It's usually best to address the `Accept` activity to the `attributedTo` or `actor` of the object, as well as any addressees the object has. For example, to accept an `Invite` to an `Event`, the `Accept` activity should include the `Invite`'s actor as well as `to`, `cc`, and `audience`.
To consume	• This activity type is usually important for updating state on an object, such as the list of people who have RSVP'd to an event. • You can use this activity to update cached state for an object you don't manage, but since you might not receive all `Accept` activities for that object, updating the cache directly from the sender may be wise.
See also	• `Reject`, which is the opposite of `Accept` • `TentativeAccept`, which is an alternative, less conclusive activity • `Event` • `Invite` • `Offer`

Activity

Extends	`Object`
Group	Abstract, Activity
Importance	Medium
Notes	`Activity` is an important supertype for all the activity types in the Activity Vocabulary, but it's rarely used directly as a type for AS2 objects. It's mostly important for defining the main properties for its subtypes. One way it is used directly is when multityping with extension properties, to signify that the extension type is also a kind of `Activity`. For example, `"type": ["ex:Ski", "Activity"]` indicates that the object represents an activity (here, a skiing activity), and that properties like `actor` and `object` should be used to reflect the same meanings as for other activities.
Properties	`actor`, `object`, `target`, `origin`, `result`, `instrument`
To publish	• Don't use the `Activity` type by itself. • Use each `Activity` with extension types to indicate that they are also activities.
To consume	If an object has multiple `type` values and one is `Activity`, treat the object as an activity.
See also	`IntransitiveActivity`, a useful subtype for activities that don't have an `object` property

Add

Extends	`Activity`
Group	Activity, Collection Management
Importance	High

Notes

The primary use of `Add` is to add an element to a `Collection` or `OrderedCollection`. The element being added is the *object*, and the collection being added to is the *target*.

There's no way to specify *where* in the collection the object will go. For an unordered collection, the location doesn't usually matter. For an ordered collection, the natural ordering of the collection should define where the object ends up. That said, for ActivityPub, which usually orders collections in reverse-chronological order, adding an element will make it the first element in the collection.

It's possible for an actor to add an object they don't own to a collection—for example, adding an `Article` written by someone else to a `Read later` collection. This kind of curation is important in many social applications. This doesn't usually require consent from another actor.

An actor also can add an object to a collection it doesn't own—for example, adding an `Image` to a shared `Party Photos` collection. This can require consent via an `Accept` activity.

A secondary use of this activity type is to add a `Person` to a `Group` or `Organization`. `Invite` is probably a better type for this, though, unless the acceptance of the member happens outside of ActivityPub.

Properties

`actor`, `object`, `target`

To publish

- If you have an `origin` for the element (for example, a collection it was part of before) use the `Move` activity instead.
- You can `Add` more than one `object` to a collection at a time, but not all consumers will be expecting multiple values in the `object` property. An alternative is to `Add` each object in its own activity, although this can cause a lot of extra activities. Not an easy trade-off!
- Don't add the same object to multiple `target` values.

To consume

- `Add` can be useful for caching the contents of a collection, but your processor might not always receive all modifications to the collection.
- Pay attention to multiple values in `object`.

See also

- `Remove`, which is the opposite of `Add`
- `Accept` and `Reject` for managed collections
- `Invite`

Announce

Extends	`Activity`
Group	ActivityPub, Activity, Meta-activities
Importance	High
Notes	`Announce` is the type used for sharing, reposting, or boosting an object. Besides `Create`, it may be the most important activity type. Its intent is to bring the attention of its addressees to the object. The `object` of the activity is what is being shared or announced. This is usually a content object, like a `Note`, `Article`, `Video`, `Audio`, `Page`, or `Document`. It can also be an activity, either by the actor or by another actor. Other types for the `object` are rarer.
Properties	`actor`, `object`
To publish	For maximum interoperability, use `Announce` only for sharing other activities or for content types.
To consume	`Announce` activities can be added to the `shared` property of the `object`.
See also	• `Add` for a similar curation type • `Flag` for content or activities that deserve less or no attention

Application

Extends	`Object`
Group	Actor
Importance	Medium
Notes	The `Application` type represents any software program, but usually it's specific to client applications for the ActivityPub API. It can be used as the `instrument` of an activity, to show that the activity was executed using that application. `Application` is also found in the `generator` property for a content object, such as an image or video. An `Application` can also be the actor of an activity. In ActivityPub, an actor can be an `Application`; in social networks, this kind of automated account is sometimes called a *bot*.
Properties	`name`, `summary`, `icon`, `attributedTo`, `image`, `preview`

To publish	Use `Application` for the following: • The `instrument` of an activity • The `generator` of a `content` type object • The `actor` of an activity that operates automatically
To consume	Treat `Application` objects like other actor types. Some publishers use this type for the server actor.
See also	`Service` for server software

Arrive

Extends	`IntransitiveActivity`
Group	Geosocial, Activity, Intransitive
Importance	Low
Notes	`Arrive` defines the actor arriving at the `location` (usually a `Place`), and `origin` is used to show where the actor is arriving from, if anywhere. Note that the other end of the line isn't `target`, but `location`. `Arrival` is an instantaneous event, without a `startTime`, `endTime`, or `duration`. Descriptions, reviews, or photos make sense as `attachments` for the `Arrive` type. However, if the content isn't created on site—that is, the actor isn't actually at the `location`—it may make more sense to use a `Create` activity with a `Note` or `Image` that includes a `tag` property with the `Place` as a value. This type is used for geosocial "check-in" activities, and it's important in that domain. However, as I write this in early 2024, geosocial activities aren't a big part of the fediverse (which is too bad, because they're really fun!). I hope that use of `Arrive` will increase in the future. Unlike other common English verbs in the Activity Vocabulary, the description of `Arrive` is fairly specific about arriving at physical locations. So avoid connotative uses of `Arrive`, like arriving at a decision, a conclusion, or a web page.
Properties	`actor`, `location`, `origin`, `attachments`

To publish
- Use a `location` with a `Place` type.
- Use `attachments` for any content from the actor.
- Since this activity type is not widely used, include a `summary` for consumers to use as a fallback representation.
- Take care in sharing or publishing specific, current location information for a person. It can be a serious safety issue. By default, restrict its distribution to followers or other well-known people.

To consume Some social software updates the `location` of the actor object when a geosocial activity is received.

See also
- `Leave`
- `Travel`
- `Place`
- `location`

Article

Extends `Object`
Group Object, Content Object
Importance High

Notes

`Article` is the type for multiparagraph written works like magazine articles, blog posts, wiki pages, or scientific journal articles. (That doesn't usually mean book-length or encyclopedia-length: especially for delivery via ActivityPub, try to keep `Article` lengths to a deliverable size—maybe 5,000 words maximum.)

The text goes in the `content` property.

An `Article` is distinguished from a `Note` by its length. It also usually has a title; for AS2, that's provided by the `name` property.

The lede, abstract, or other introductory information should be in the `summary`.

The `content` should be HTML5. It's possible to use the `mediaType` property to define another `content` type, but there's no guarantee consumers will know how to process that type.

Consumers should sanitize the content before showing it to users, especially in a web browser. Running arbitrary HTML5, with scripts, stylesheets, embedded objects, or other executable code in a user's browser environment is a security issue.

Embedded images that require authorization might not be processed correctly. If the layout of the article makes it possible, images may make more sense as `attachments` displayed outside the text flow.

The text's author goes into `attributedTo`, and `published` and `updated` can give publication and other dates. The Activity Vocabulary doesn't have properties for other roles like publisher, editor, or photographer. Nor are there properties to indicate the publication's name or issue.

Properties

`attributedTo`, `content`, `name`, `summary`, `attachments`

To publish

- Use `content` for multiparagraph HTML5 content.
- Use `summary` for an introduction to the content of a single paragraph or less.
- Use `name` for the title.
- Use `attachments` for images or other visual presentations.
- Keep the length under 5,000 words.
- Don't include executable JavaScript or CSS stylesheets.

To consume

- Sanitize the content before displaying it to a user, especially in a web browser. Use an HTML parser or a dedicated package to remove JavaScript, CSS, and other elements and attributes that can cause problems with the user's display.
- Presenting the `name` and `summary` in a feed layout can keep the presentation about the same size as other content or activities. A Read More or similar button can be used to open up the full contents, either in-stream or out-of-stream.
- If no `summary` is provided, you can use the first paragraph of the `content` property for that purpose.

See also

- `Document` for longer, binary-format, or structured texts
- `Note`
- `Page`, for an HTML web page included by URL
- Dublin Core, for additional publication metadata

Audio

Extends `Object`

Group Object, Content Object

Importance Medium

Notes The `Audio` type is used for any audio content, such as music, conversations, voice memos, podcasts, and sound effects. Two types of audio matter here:

- Audio created within the ActivityPub network, by uploading or recording audio files; these are relatively easy to assign an ID and URL.
- Audio created elsewhere, like a National Public Radio (NPR) podcast or the song "Ring of Fire" by Johnny Cash. Referring to these audio objects with, say, a `Listen` activity is more difficult, since there's not an easy ID for them. You have two options: leave out the ID or use a custom ID. Neither is great.

There is no good property to use for a larger album, channel, or podcast the audio is "part of." The `partOf` property is just for `collection` pages, but `context` provides a good fallback.

Properties `url`, `name`, `attributedTo`, `mediaType`, `context`

To publish
- Use `url` for an external link to the audio object.
- Use `name` for the title.
- Use `attributedTo` for the author information. This can be a `Link` if the author does not have an ActivityPub ID.
- Use `mediaType` if you know it.
- Use `context` for the album, podcast, or channel to which the audio belongs.

To consume Don't depend on having a unique `id` across services.

See also
- `Listen`, for the experience activity related to `Audio`
- `Link`, for external data

Block

Extends `Ignore`

Group Activity, ActivityPub, Moderation Activity

Importance High

Notes `Block` is used for blocking other actors in the ActivityPub specification. The `object` property is the actor being blocked. Blocked actors cannot read the actor's data, activities, or created content or interact with the blocker, including replies, likes, and shares. Typically, any follower-following relationship also ends.

ActivityPub requires that this activity type not be propagated to the blocked actor. It's principally important for the ActivityPub API, as a way for the user to say whom they're blocking. It would be unusual for any consumer besides the user's ActivityPub API server to receive this activity. An exception might be for shared blocklists, in order to collate frequently blocked users.

The `blocks` property is made up of `Block` activities, which can be used to `Undo` the activity.

Properties `actor`, `object`

To publish Only the `object` is important for this type.

To consume This is an unusual type for any consumer except the actor's ActivityPub API server.

See also
- `Ignore`, for a milder form
- `Undo`, to remove a block

Collection

Extends	`Object`
Group	Core, Collection
Importance	Medium
Notes	The `Collection` type is the base type for all the collection types. It can be used for feeds in ActivityPub, but `OrderedCollection` is clearer for those uses. This type has two main variants: a small collection, in which the members of the collection are included in the `items` property, and a paged collection, in which the `first` or `last` properties point to "pages" of results, each of which has its own items.
Properties	`totalItems`, `items`, `first`, `last`, `current`
To publish	• Avoid this type and use `OrderedCollection` if the collection should be ordered. Most important collections in ActivityPub are ordered! • Use paging for large collections (more than 20 or so); include an `items` property for small collections. Use paging for potentially large collections; for example, the streams defined in ActivityPub should be paged, even if they have no members. • Include `totalItems` if possible. It's much easier for you to calculate than for consumers! • Use paging, starting from the `first` property and following the `next` property. • If possible, also include `last` and `prev` properties. • Use only the `CollectionPage` type for pages.

To consume

- The `Collection` type may be used in contexts in ActivityPub where the collection should be ordered—for example, most streams. In those cases, treat the content as ordered.
- The `totalItems` property may not reflect the total number of items the user can actually read. Take care not to use this property as a fixed size for the collection (for example, in allocating memory for an array).
- The `first` and `last` pages of a collection can change over time as items are added.
- This type can have an `orderedItems` property; check for it if `items` is missing.
- If there is a single element in `items` or `orderedItems`, some processors will publish it as a single object, not an array with a single element. (This is an artifact of JSON-LD canonicalization.) Be prepared for nonarray values for these properties!

See also `OrderedCollection`, for a collection in which order matters

CollectionPage

Extends `Collection`

Group Collection

Importance Medium

Notes A `CollectionPage` is a subset of a collection. Much like a linked list, it uses a `next` or `prev` property to point to the next page of items. The `partOf` property identifies the collection of which this page is a part.

Properties `items`, `next`, `prev`, `partOf`

To publish

- `CollectionPage` should be used only as a page in a `Collection` type. If the collection is ordered, use `OrderedCollectionPage` instead.
- Include a `next` property if possible.
- Include a `partOf` property if possible.
- The `prev` property is for two-way scrolling through a collection.
- Use `items` for the items.
- Don't use `totalItems` for pages.

To consume
- Some publishers use this type as a page of an `OrderedCollection`.
- Some publishers use `orderedItems`; check for it if `items` is missing.
- If there is a single element in `items` or `orderedItems`, some processors will publish it as a single object, not an array with a single element. (This is an artifact of JSON-LD canonicalization.) Be prepared for nonarray values for these properties!

See also
- `OrderedCollectionPage`, preferred for the ordered collections in ActivityPub
- `Collection`

Create

Extends `Activity`

Group Activity, CRUD Activity, ActivityPub

Importance High

Notes The vast majority of activities on the ActivityPub network are `Create` activities. This is the type used for creating new content objects, and that's what people are mostly doing on the fediverse.

Properties `actor`, `object`, `summary`

To publish
- Use `object` to represent the created object.
- Include a `summary` or `summaryMap` to clarify the effect of the activity.

To consume
- Some documentation recommends using `result` for the created object. If `object` is missing, check that property for the possible result.
- This activity is often used for managing the `replies` collection for an object. If a consumer finds a `Create` activity with a new `object` and an `inReplyTo` property representing an object it manages, it should add the new object to the `replies` collection for that object.
- A created object, especially an older one, may have been subsequently updated or deleted.

See also
- `Update`, for additional editing
- `Delete`, for deleting a created object

Delete

Extends	`Activity`
Group	Activity, CRUD Activity
Importance	High
Notes	Deleting an `object` should leave a `Tombstone` where the object once was. This makes it unnecessary to find and remove the object from the many collections or activities it may be a member of. Instead, when the collection is being shown in a user interface, the `Tombstone` objects can be skipped. Deletions can be reversible; it may make sense to move deleted objects to separate storage so they can be restored with an `Undo` activity. That storage should be eventually emptied to respect the user's preference for permanent deletion.
Properties	`actor`, `object`
To publish	• Use a `Tombstone` type for the object, with the `formerType` set to the former type of the object. • Although the `object` property can be an array, some consumers will not be prepared for multiple values. If needed, use multiple `Delete` activities, each with one `object`. • Avoid using the `origin` property to delete items from a collection, which is rare. Instead, use a separate `Remove` activity.
To consume	• Removing objects from caches or other storage is important for respecting authors' choices. • A `Delete` activity can have an array in its `object` property, consisting of each item that has been deleted. • The `origin` property can be used to simultaneously remove the object from a collection. If it's present, treat it as an additional `Remove` activity.
See also	• `Create`, for creating an object • `Tombstone`, for the result of the deletion • `Undo` • `Remove`, for a better way to remove an item from a collection

Dislike

Extends	`Activity`
Group	Activity, Reaction Activity
Importance	Low
Notes	`Dislike` indicates that the `actor` dislikes the `object`. Note that there is a distinction between dislike and ambivalence. The `Dislike` activity type is unusual on the fediverse. It's mostly useful as a downvote signal. Unlike the `Like` activity, `Dislikes` of an object are not tracked in a property of that object.
Properties	`actor`, `object`
To publish	Use a separate `Undo` activity to undo a `Like` activity, rather than a `Dislike`.
To consume	N/A
See also	• `Like` • `Undo`

Document

Extends	`Object`
Group	Content Object
Importance	Medium
Notes	Although `Document` can represent any kind of document, it's mostly used for binary file formats, such as PDFs, Google Docs, or Microsoft Office documents. `Documents` can be published on their own or as part of the `attachments` array of another object. The contents of the `Document` are usually at the URL represented by `url` and are not included in the `content` property.
Properties	`url`, `name`, `mediaType`, `summary`, `attributedTo`, `icon`, `image`, `published`, `updated`

To publish

- Providing metadata about the `Document` can save consumers the time and bandwidth of downloading it.
- Use `url` for the location of the document.
- Use `name` for the filename or title.
- Use `mediaType` for the document type.
- Use `summary` for a brief description of the contents of the document.
- Use `attributedTo` for the author of the document.
- Use `icon` to represent the type of file.
- The `image` should be a screenshot or excerpt.

To consume

- Check for necessary metadata about the document in the JSON representation before fetching the document URL. You may have sufficient information to represent it already, without downloading it.
- If `mediaType` is not provided, you can often infer the type by using the "file extension" in the URL.
- As a fallback, use `mediaType` to identify a likely `icon` representation.

See also

- `Article`, for an HTML document with contents inline
- `Page`, for a web page
- `Video`, for a more specific type for video files
- `Audio`, for a more specific type for audio files
- `Image`, for a more specific type for image files

Event

Extends `Object`
Group Real World
Importance Medium

Notes	Although `Event` can represent any kind of event (like the Battle of Waterloo or the first moon landing), it is most often used in social settings, for meetings and get-togethers. As with many types, the addressing properties (`to`, `cc`, and so on) set the read-only scope for the `Event`. Whether additional `Invite` activities expand this list is implementation-dependent. The addressing properties can be `Collections`, `groups`, `organizations`, or the `Public`.
Properties	`name`, `summary`, `startTime`, `endTime`, `duration`, `location`, `attributedTo`
To publish	• Include both `startTime` and `endTime`. • The `location` should be a `Place`. • The value for `attributedTo` should be the creator and manager of the event, not necessarily a performer or speaker. • The `summary` should be a description of the event.
To consume	If `endTime` is missing, look for `duration` and use that to calculate the end time.
See also	`Invite`, for inviting actors to an `Event`

Flag

Extends	`Activity`
Group	Moderation Activity
Importance	High
Notes	The `Flag` type is used for highlighting problematic content or actors. The `object` is the object of concern. It can be an actor object, like a `Person`, or a content object, like a `Note`. These reports are usually reviewed by the moderators of the user's server, as well as moderators of the flagged person's server.
Properties	`actor`, `object`, `summary`
To publish	The `Flag` activity should not be addressed to the person of concern or the author of the content of concern.
To consume	`Flag` usually has the side effect of opening an issue or report on multiple servers. The visibility of this report may not reflect the addressing properties of the `Flag`.
See also	`Block`, for blocking a problematic user

Follow

Extends	`Activity`
Group	Activity, Social Graph Activity, ActivityPub
Importance	High
Notes	The `Follow` activity type is essential for creating the directed social graph used in ActivityPub. See Chapters 3 and 4 for explanations of how the type is used.
Properties	`actor`, `object`
To publish	This type is typically addressed only to the followed actor.
To consume	When this type is received, an `Accept` or `Reject` activity is usually generated, often due to input from the user.
See also	• `Accept` • `Reject` • `Undo` • `Relationship`

Group

Extends	`Object`
Group	Actor
Importance	Medium
Notes	Although it can represent any type of group, `Group` is most often used for online communities, like a Facebook Group, a Slack channel, or a Subreddit. Content addressed to a `Group` actor by one of its followers is sometimes re-shared to all the group's followers. See FEP-1b12 for one way to do public group sharing.
Properties	`name`, `summary`, `icon`, `image`, `inbox`, `outbox`, `followers`
To publish	• `name` is the name of the group. • `summary` is a brief description of the group. • `icon` is a group avatar. • `image` is often used as a profile page background. • `inbox`, `outbox`, and `followers` are useful if the group can be followed.

To consume	N/A
See also	• `Join`, for joining a group • `Leave`, for leaving a group • `Follow`, for a looser membership in a group

Ignore

Extends	`Activity`
Group	Reaction Activity, Moderation Activity
Importance	Medium
Notes	The `Ignore` activity type implements the "mute" pattern in social networks. If an actor indicates they want to ignore an object or actor, they should not see further activities about the object or by the actor, including reactions and replies. If the ignored object is an actor, the ignored actor can still follow the ignoring actor, and can read, reply to, and react to their content. The actor just won't *see* any of these activities. If the ignored object is a content object, like a `Note` or an `Image`, the ignoring actor will typically not see further reactions or replies to that object, nor to replies in the same thread.
Properties	`actor`, `object`
To publish	Typically, this type is not delivered to the ignored actor or the author of an ignored object.
To consume	N/A
See also	`Block`, for a stronger moderation action

Image

Extends	`Document`
Group	Content Object
Importance	High

Notes This is the most important and widely used content type, with the possible exception of `Note`. The `Image` type is used for any two-dimensional image, but primarily for digital images in web-friendly formats like JPEG, WebP, PNG, and GIF.

Image types are useful as content on their own but are also important as the `image` or `icon` property of another object.

Some systems, like Mastodon, include images as `attachments` to another object.

Despite their obvious utility, the `width` and `height` properties don't apply for this type.

Note that `id` should be the URL of the JSON representation of the image, and `url` should be the URL of the binary data.

Properties `name`, `url`, `attributedTo`, `summary`, `mediaType`, `to`, `cc`

To publish

- `url` provides access to the content of the image file.
- `mediaType` gives a hint on the file type.
- `name` can be used in place of the image or as a title.
- `attributedTo` describes the creator.
- `summary` provides useful alternative text.
- `to`, `cc` show who has read-only access to the image.
- Avoid `width` and `height`, since they aren't defined for this type.

To consume

- If the `mediaType` is not available, it may make sense to guess the type from the URL's extension (for example, *.jpg*)
- You can also get the type when it's returned by an HTTP server.
- `width` and `height` are nonstandard for this type but occasionally provided.

See also `Link`: specifically for links without IDs; includes width and height properties

IntransitiveActivity

Extends `Activity`
Group Activity, Abstract, Core
Importance Low

Notes	`IntransitiveActivity` is an abstract type for activities that don't have an `object` property. There are three in the Activity Vocabulary: `Arrive`, `Travel`, and `Question`. It can also be used as the base type for an extension activity type, if it's important to indicate that the activity doesn't have an object.
Properties	`actor`, `summary`, `target`, `origin`, `result`, `location`
To publish	This type can be used for multityping, but otherwise, you shouldn't publish any objects with type `IntransitiveActivity`. Pick a more specific type from the Activity Vocabulary or develop an extension.
To consume	`summary` can provide an overview of what this object is supposed to represent.
See also	`Activity`

Invite

Extends	`Offer`
Group	Activity
Importance	Medium
Notes	The format of this activity is a little strange; normally you'd invite a person to an event, but in this case you invite the event to the person. The reason is that `Invite` extends `Offer`, and we usually offer an object to a person. `Event` is by far the most important possible `object` type for this activity, followed by `Group`. The `Relationship` schema defined in the Activity Vocabulary uses `Invite`, but it's not widely implemented. An `Organization` might also be an object for `Invite`; so might a `Place`, a `Service`, or an `Application`. After that, it's hard to see meaningful use for this type for other types in the vocabulary. The `target` of the object is who the invitation is for; usually this is a `Person`, but it could also be another actor type.
Properties	`actor`, `object`, `target`
To publish	• `object` is the event or other entity. • `target` is the list of actors being invited to the event. • Include all the actors in the addressing properties.
To consume	`target` can have multiple values.

See also

- `Accept`
- `Reject`
- `TentativeAccept`
- `TentativeReject`
- `Ignore`

Join

Extends `Activity`

Group Activity, Group

Importance Medium

Notes The `Join` type is used when someone joins something. The `object` is what was joined.

This is useful for joining a `Group`. It can also be used for joining a `Service` or an `Organization`.

Properties `actor`, `object`

To publish The `object` is the thing being joined. Avoid using multiple values here.

To consume Some `objects` may require approval to join.

See also

- `Leave`, the opposite of `Join`
- `Follow`, sometimes used as an alternative for joining a group

Leave

Extends `Activity`

Group Activity, Geosocial, Group

Importance Medium

Notes	`Leave` is a useful type for when someone leaves a thing. It's specifically defined for two types of object: `Group` and `Place`. For `Group`, it is used for when someone leaves a group. For `Place`, `Leave` is a geosocial event, like a check-in, but for leaving the place. Some vulnerable populations prefer to check in when they leave a place, instead of when arriving, to avoid having anyone find them at the place while they're there. Other types in the Activity Vocabulary that might be useful as the `object` are `Organization` and `Service`.
Properties	`actor`, `object`
To publish	Avoid using anything but a `Place` or `Group` as the `object` type. Use `endTime` to indicate when the actor has completed leaving.
To consume	The `object` type may not be a `Place` or `Group`. Use `endTime` to indicate when the event occurred; `published` can be used as a fallback.
See also	• `Join`, the opposite of `Leave` • `Undo` • `Group`

Like

Extends	`Activity`
Group	ActivityPub, Reaction Activity
Importance	High
Notes	The `Like` activity is a positive reaction that says that the actor "liked" the `object`. In some systems, this is called an *upvote*, *favorite*, *+1*, or *thumbs-up*. User interfaces often represent this activity with a heart or star. `Like` is usually treated as idempotent; liking something multiple times is the same as liking it once. Unlike the Like with Facebook Pages, `Like` in ActivityPub does not usually subscribe to updates by the object. This is the most important and most widely implemented activity outside of the content management types (`Create`, `Update`, `Delete`).
Properties	`actor`, `object`

To publish
- Content types like `Note`, `Image`, or `Video` are widely supported as the object of a `Like`.
- Activity types like `Follow` or `Arrive` are somewhat widely supported, but less so.
- Actor types as the object, like `Person` or `Organization`, are rarely supported.

To consume If a processor sees a `Like` activity for an object the processor manages, it should add the activity to the object's `likes` property, if it exists.

See also
- `Dislike` for downvoting or disliking an object
- `Undo` for undoing a `Like` activity

Link

Extends N/A

Group Core, Link

Importance Medium

Notes A `Link` is similar to the HTML `<link>` or `<a>` tags. It provides a link to a URL with some metadata about the object found at that URL, as well as the relationship with the linking object.

A `Link` is a JSON object, but it's the only core type that isn't an Activity Streams `Object`.

Note that two links with the same `href` can be different. The link includes relationship data in the `rel` property. A link with `rel` set to `next` is not the same as a link with `rel` set to `prev`.

Many object properties in the Activity Vocabulary can have a `Link` as a value, but for a few `Link` is common: `image`, for an image of the object; `icon`, for a smaller image of the object; and `preview`, for pages or videos, are often `Link` objects with an image. Similarly, `url` can also be a `Link`.

The main difference between using a `Link` or, say, an `Image` is whether there's an available AS2 representation at the object's host server. If there is, use the relevant AS2 type, since it has additional metadata like addressees, reactions, and author. If not, use the `Link` type.

Properties `href`, `name`, `mediaType`, `rel`, `width`, `height`

To publish

- Prefer `Link` objects to bare URLs as strings, even for very simple URLs.
- The `href` property contains the URL.
- The `name` is a title or name for the object.
- The `mediaType` indicates the type of resource to be found at the URL.
- The `rel` property contains the linked object's relationship to the object that has this `Link` as a property value.
- The `rel` property is informative but shouldn't contradict the object property for which the `Link` is a value.
- Including an `id` value can help consumers maintain distinct identities for `Link` objects.
- The `rel` value should be one of the link relations from IANA's registry of link relation types.

To consume

- Many properties in Activity Streams objects can have a `Link` object as their value.
- The `rel` property is informative; the property of the object for which this `Link` is a value is the more important relationship. So if a `CollectionPage` has a `next` property that is a `Link` with a `rel` value equal to `prev`, the object property takes precedence.
- `Two link` values with the same `href` property can be different. Use either the `id` or a compound of `href` and `rel`.

See also

- `Image`
- `Video`
- `Audio`
- `Page`
- `Document`

Listen

Extends	`Activity`
Group	Activity, Content Experience
Importance	Low

Notes

The `Listen` activity type is how you note that a user has listened to a piece of music or other audio. It gives a social music or podcasting experience, similar to *Last.fm*.

This activity type is not widely supported on the fediverse as of this writing, so generated activities won't be visible in applications. Use `summary` as a fallback representation.

There's not a well-agreed-upon public vocabulary for music or other audio, so `Audio` items don't have a unique ID. One option is to use a `url` property for the `Audio`; another is to use a `Link`. For audiobooks, `Read` may make more sense.

Because a `Listen` activity can happen across a duration of time, `startTime`, `endTime`, and `duration` are important for this activity.

Properties

`actor`, `object`, `summary`, `startTime`, `endTime`, `duration`, `instrument`

To publish

- Use `Audio` for audio objects that have an ActivityPub ID.
- Use `Link` if no ActivityPub ID exists.
- Provide a `summary` for fallback representation ("Evan listened to A Little Respect by Erasure"), since not all ActivityPub processors recognize this activity type.
- You can represent the activity in time by using `startTime`, `endTime`, and `duration`.
- Use `instrument` for the tool used for listening, like a streaming music service or podcatcher app.

To consume

- Use `startTime`, `endTime`, and `duration` for the time of the action, or use `published` as a fallback.
- If no `id` is provided for the `object`, you can use `url` (`Audio`) or `href` (`Link`) as a fallback.

See also

- `Audio`, for explanation of how to structure audio objects
- `Link`
- `Read`, for reading

Mention

Extends `Link`
Group Link
Importance High

Notes This type is specifically used for tagging someone in a post, such as `@sally`, when that person doesn't have an ActivityPub account.

However, Mastodon uses it for ActivityPub users too. Also, Mastodon will not deliver any content to an individual user unless that content has a `tag` with a `Mention` of that user.

Properties `href`, `name`

To publish
- `href` should be the profile URL for the mentioned account.
- `name` should be the microsyntax used for the mentioned account, like `@sally`.

To consume The `href` may be an ActivityPub actor `id`.

See also
- `Person`
- `Link`

Move

Extends `Activity`

Group Activity, Collection Management

Importance High

Notes `Move` represents moving something from one place to another. The `origin` is the place it came from; the `target` is where it's moved to. A canonical example is moving an object from one collection to another; this can also be represented with a `Remove` activity followed by an `Add` activity.

`Move` is not included in the ActivityPub specification, possibly because it can be represented with adding and removing. It is very important, however, for data portability. Mastodon uses this object type when a user moves from one account to another. The `object` is the old account, and the `target` is the new account. This usage doesn't exactly match the semantics of the `Move` activity, but it's an important part of the fediverse.

Properties `actor`, `object`, `target`, `origin`

To publish
- Use `object` for the object being moved.
- Use `target` for the container to which it is being moved.
- Use `origin` for the container from which it is being moved.

To consume A `Move` with an actor as both `object` and `target` is a Mastodon move notification. See Chapter 4 on how to handle this activity.

See also

- `Add`
- `Remove`
- `Collection`

Note

Extends `Object`

Group Object, Content Object

Importance High

Notes `Note` represents a short text, of about a paragraph or less, such as a microblogging post, tweet, or comment. It is the primary format for most microblogging services on the fediverse.

`Note` is normally self-contained, without an external `url` for content. The text of the note is in the `content` element, usually as HTML.

In a microblogging context, a `Note` may have an `attachment` property with one or more content-type objects.

Properties `content`, `attributedTo`, `attachment`, `inReplyTo`, `replies`, `likes`, `shares`

To publish

- The `content` property should include the content of the note.
- The `attributedTo` property is the author.
- For a text representation, a `Note` is usually too short to merit a title used in `name`. A `summary` would be appropriate ("A note by Evan"), but Mastodon takes a `summary` to be a content warning ("US Politics"), and blurs the content. It's acceptable to leave both `name` and `summary` undefined for a `Note`.
- If the note is a comment or a reply to another object, the `inReplyTo` property identifies the original object.

To consume

- A `Note` will not usually have a `name` or `summary`. A fallback text representation might be a combination of the type and the author's name, like "A note by Evan."
- HTML `content` should be scrubbed before including it in a web interface; it may include scripts, binary objects, CSS, or other security-sensitive content.
- The `replies`, `likes`, and `shares` properties are helpful for identifying reactions to the content.

See also

- `Hashtag`
- `Mention`
- `Article`
- `Question`

Object

Extends N/A

Group Core

Importance Medium

Notes This is the supertype for every other type in the Activity Vocabulary, except `Link` and `Mention`.

Using `Object` as a type directly is unusual; usually, you'll use more particular types instead. However, it may occasionally be used for a multityped object with an extension type, to show that the object is based on AS2 and can use `Object` properties.

Otherwise, its use indicates that the publisher isn't sure what type of object it is, and is just giving the most generic possible properties and types.

Properties `id`, `name`, `summary`, `icon`, `image`

To publish

- Avoid using this type directly, except for multityping.
- Because this type is so generic, if you use it, make sure to include fallback text representations in `name` or `summary` and in visual representation as `icon` or `image`.

To consume This object type has little context, so fallback text representation (`name`, `summary`) and image representations (`icon`, `image`) are really the only clues to go on.

See also `Link`

Offer

Extends `Activity`

Group Activity, Relationship

Importance Medium

Notes `Offer` is an `Activity` type for when the actor offers something (the `object`) to someone (the `target`).

Many use cases for `Offer` are covered better by the more specific activity type `Invite`, especially those that involve joining or attending something.

The main case covered in the Activity Vocabulary is for offering a `Relationship` in order to initiate a two-way connection. This is an important use case, but much of the fediverse leans toward the one-way connections encoded in ActivityPub's follower/following structure.

The other place where an `Offer` makes sense instead of an `Invite` is for commerce—offering an object for sale or auction, or offering a discount. Unfortunately, there isn't much other vocabulary for ecommerce in AS2, so it's hard to fit this type into a commercial structure. I hope it gets used for a future extension.

Properties `actor`, `object`, `target`

To publish If published over ActivityPub, also include the `target` as an addressee in the `to` property.

To consume An `Offer` activity should usually generate an `Accept` or `Reject` activity.

See also

- `Invite`
- `Relationship`
- `Follow`

OrderedCollection

Extends `Collection`

Group Collection

Importance High

Notes `OrderedCollection` objects represent a collection of objects where order matters.

Most of the important collections in ActivityPub are ordered in reverse-chronological order, which means they should be `OrderedCollection` objects.

Small `OrderedCollection` objects with a fixed set of contents should include an `orderedItems` array with the contents. Larger ordered collections should be broken into pages.

Properties `first`, `totalItems`, `last`, `current`

To publish

- If the collection is small (~20 items or less) and will not change, use the `orderedItems` property to represent the items.
- Otherwise, include a `first` property whose value is an `OrderedCollectionPage`.
- If known, a `last` property can help users navigate the collection backward as well as forward.
- For collections in reverse-chronological order, the `first` page contains the newest items, and the `last` page contains the oldest items.
- Including a `current` with the page with the newest items (equal to `first` if the collection is reverse chronological) can help establish the order of the collection.

To consume

- For collections in reverse-chronological order, the `first` page contains the newest items, and the `last` page contains the oldest items.
- If the `OrderedCollection` has an `orderedItems` member, this will be the full contents of the collection.
- If it has a `first` property, you can navigate the entire collection by starting with the `first` page and following each page's `next` property.
- If the `OrderedCollection` has a `last` property, you can navigate the entire collection by starting with the `last` page and following each `prev` property.
- You can have an `OrderedCollection` with `CollectionPage` objects as pages.

See also

- `OrderedCollectionPage`, for pages
- `Collection`, usually for unsorted collections

OrderedCollectionPage

Extends	`CollectionPage`
Group	Collection
Importance	High
Notes	This type represents a segment of an `OrderedCollection`. Usually, the pages are distinct; no item is present in more than one page.
Properties	`orderedItems`, `next`, `partOf`, `prev`

To publish
- This type should be used for the pages for `OrderedCollection` objects only.
- Include a `next` property if possible.
- Include a `partOf` property if possible.
- The `prev` property is for two-way scrolling through a collection.
- Use `items` for the items.
- Don't use `totalItems` for pages.

To consume
- Some publishers use this type as a page of a `Collection`.
- Some publishers use `items` for the items; check for it if `orderedItems` is missing.
- If there is a single element in `items` or `orderedItems`, some processors will publish it as a single object, not an array with a single element. (This is an artifact of JSON-LD canonicalization.) Be prepared for nonarray values for these properties!

See also
- `OrderedCollection`, for the container type
- `CollectionPage`, for the unordered page type

Organization

Extends `Object`

Group Actor

Importance Medium

Notes The `Organization` type is used for organizations that exist offline, such as a company, university, government, nonprofit, union, club, or other group of people.

Membership in an `Organization` is usually handled out of band—not online, as with a `Group`.

`Organization` objects can be actors and are reasonably well supported.

Properties `name`, `summary`, `icon`, `url`

To publish
- Use `name` for the name of the organization.
- Use `summary` for a description of the organization.
- Use `icon` for a small, square image of the organization, like a logo.
- Use `url` for the home page of the organization.

To consume `Organization` is a type sometimes found for actor objects.

See also `Group`, for online groups

Page

Extends	`Document`
Group	Content Object
Importance	Medium
Notes	The `Page` type represents a web page—roughly, what's linked to by a URL. It's especially useful for sharing links to web pages, either in an `Announce` activity or as a `tag` in a `Note`. This type is distinct from a `Link` in that it includes metadata about the linked page.
Properties	`url`, `name`, `summary`, `image`, `icon`, `attributedTo`
To publish	• Use `url` for the URL of the web page. • Use `name` for the title of the web page. • Use `summary` for a description of the web page. • Use `image` for a snapshot of the web page or a representative image from the page. • Use `icon` for a small, square image representing the web page (often a logo). • Use `attributedTo` for the author of the page.
To consume	Watch for multiple values in `image`, `icon`, and `attributedTo`.
See also	• `Application`, for web applications • `Profile`, specifically for profile pages • `Link`, for links to web resources without metadata

Person

Extends	`Object`
Group	Actor
Importance	High
Notes	The `Person` type represents a person, real or fictional. It's most commonly used for user accounts on the ActivityPub network.
Properties	`name`, `preferredUsername`, `icon`, `summary`, `location`

To publish	• Use `preferredUsername` for the username of the person, if applicable. • Use `name` for their full name. • Use `icon` for an avatar photo. • Use `summary` for a description, such as a bio. • Use `location` for a location the person is now, or where they are usually.
To consume	• Some systems use `Person` for all user accounts, whether they are for organizations, bots, or other entities. • Some publishers use `content` for the bio or description.
See also	• `Application`, for automated user accounts (bots) • `Organization`, for a real-world group

Place

Extends	`Object`
Group	Geosocial, Real World, Object
Importance	Medium
Notes	The `Place` type represents a place in the world. It's primarily used for real places but can also be used for fictional ones. It's primarily structured for point-like places—businesses, parks, even towns or cities that can be represented with a single latitude/longitude coordinate pair. Bigger places, like regions, states, and countries, can be identified by name and ID. There is no easy way to represent a polygon for a place. The Schema.org vocabulary provides a `Place` type with some additional properties. Using the Schema.org version of `Place` with the AS2 `Place` may be a useful way to include more details about the place. This type is primarily used for the `location` property of other objects and activities. It can also be used for geosocial activities like `Arrive`, `Leave`, and `Travel`. However, this property is not widely supported, nor are geosocial activities.
Properties	`name`, `summary`, `latitude`, `longitude`, `image`, `altitude`

To publish	• Use `name` or `nameMap` for the name of the place. • Use `summary` for a description of the place. • Use `latitude` and `longitude` for the coordinates of the place. • Use `image` for an illustrative image of the place. • Avoid `accuracy`, `radius`, and `units` as they are poorly defined (by me!) and not well understood.
To consume	It can be hard to match a `Place` to an actual place without a good ID vocabulary or accurate latitude/longitude values. Many places have the same name (for instance, 50 US cities are named Springfield).
See also	• Schema.org `Place` • GeoJSON-LD, for additional information about the shape of a place

Profile

Extends	`Object`
Group	Object
Importance	Low
Notes	This type represents a profile page for a person or other actor. Publishers should probably avoid this type in favor of a `Link` value for the `url` property of the object.
Properties	`url`, `describes`
To publish	• Use `url` to share the profile URL. • Use `describes` for the object described by the profile.
To consume	Usually, the necessary information is in the `describes` property.
See also	• `Page`, for any web page • `Person`, for describing a person directly

Question

Extends	`Activity`
Group	Question, Activity
Importance	High

Notes

The `Question` type is used for asking open-ended questions or for polls that offer a limited set of answers.

Most activity types in the Activity Vocabulary map onto an English verb, like `Delete` or `Like`. `Question` may seem like an odd choice for an activity type, but it is also an English verb.

For open-ended questions, the answers are in the `replies` collection.

For multiple-choice questions, the possible answers are in the `anyOf` or `oneOf` properties.

Multiple-choice answers can be of any type, but `Note` or `Object` are the most common.

Polls are widely supported on the fediverse.

Properties

`content`, `oneOf`, `anyOf`, `endTime`, `closed`, `replies`, `result`

To publish

- Use `content` or `contentMap` for the text of the question.
- Use `oneOf` for multiple-choice polls with exclusive answers.
- Use `anyOf` for multiple-choice polls where more than one answer can be returned.
- Use `endTime` to show when a question will end, if it is time-limited.
- Use `closed` for the timestamp when the question closed.
- Use `replies` to show all answers to the `Question`, as well as other comments or replies.
- Use `result` to show which answer was chosen, either multiple-choice or open-ended.

To consume

- The `closed` property can include multiple types, such as Booleans, datetimes, or objects.
- Replies to multiple-choice questions should include the `Question`'s ID in the `inReplyTo` property and should match the name of the choice offered.
- Although not strictly correct, some implementations will include a `replies` property on each answer in a multiple-choice question, with `totalItems` showing the count of replies that match that answer.

See also

- `Article`
- `Note`

Read

Extends	`Activity`
Group	Content Experience, Activity
Importance	Medium
Notes	The `Read` type indicates that the actor has read an object. It's most useful when a `Person` has read some textual content, like a `Note`, `Article`, or `Page`. The `object` can also be a `Link`, for when the textual content does not have an AS2 representation. `Read` activities usually have an extent in time (reading a book can take weeks or months, for example). The `duration`, `startTime`, and `endTime` properties are all useful for this activity type. Although this activity is useful for lots of kinds of content, it is not widely implemented on the fediverse.
Properties	`actor`, `object`, `duration`, `startTime`, `endTime`, `result`
To publish	• Use `object` for the object that was read. • Use `duration`, `startTime`, and/or `endTime` to show how long the reading took and when it started and ended. • Use `result` for any result of reading the text, such as notes or a review.
To consume	N/A
See also	• `View` • `Listen`

Reject

Extends	`Activity`
Group	ActivityPub, Event Management, Meta-activities
Importance	High

Notes

The `Reject` type is for rejecting an object or activity, usually when that object has been presented for some kind of approval by the actor.

This type is most important for rejecting a `Follow` activity in ActivityPub. It can also be used for rejecting an `Offer` or `Invite` activity.

Rejecting other kinds of objects is less well-defined. Rejecting an `Event` is implicitly rejecting an invitation to that event.

Some implementations also use `Reject` to signify that they are not going to process an `Activity` because they don't support it: for example, an implementation that receives an `Offer` for a `Relationship` that doesn't support two-way following.

It's possible to reject an object that has been submitted for inclusion in a `Collection`, in which case the `target` should be the relevant collection. This parallels how `Accept` handles collections.

Properties `object`, `target`

To publish

- Avoid using `Reject` for rejecting nonactivity objects, since they're not well-defined.
- Use `object` to represent the activity being rejected.
- If the object is being rejected from inclusion in a `Collection`, use `target` for the collection.

To consume

- If the `object` is a `Follow`, `Offer`, or `Invite`, you can assume that the actor has understood and rejected the activity.
- Some processors send a `Reject` activity if they will not process the `object`.

See also

- `Accept`, the opposite of `Reject`
- `Dislike`, for when approval is not solicited
- `Remove`, for removing an object from a collection

Relationship

Extends `Object`
Group Relationship, Real World
Importance Medium

Notes

The `Relationship` type represents a relationship between two objects.

Usually, this type is for qualifying the relationship between two `Person` objects. The `relationship` property's value is a term that defines the relationship. The Activity Vocabulary does not define relationship types, but the Relationship Vocabulary is a good place to start. (I'll use the prefix `rel:` to indicate the Relationship Vocabulary in these notes.)

Relationships are not necessarily symmetrical. If the subject is the parent of the object (`rel:parentOf`), that does not mean the object is also the parent of the subject. Some are symmetrical, though: if the subject is the friend of the object (`rel:friendOf`), then the object is also the friend of the subject.

With the `Relationship` type, the `subject` is the person originating the relationship, and the `object` is the person who is the topic of the relationship. So, with the `rel:childOf` relationship, the `subject` would be the child, and the `object` would be the parent.

This type could be used to represent a relationship between a `Person` and a `Group` or `Organization`.

Relationships between other types of objects are possible but should probably be represented by more-specific properties, unless the `Relationship` itself needs to be represented as its own object. So, the relationship between a `Person` and a `Place` is better represented with a `location` property than a `Relationship` object.

The primary use of this type is for establishing relationships between ActivityPub actors. Section 5.2 in the Activity Vocabulary specification covers how to use this type for establishing a new relationship.

However, the primary way that the social graph is managed in the ActivityPub world is through the `Follow` activity, and properties of actors like `followers` and `following`. The schema described in the Activity Vocabulary is not widely used.

Properties

`subject`, `object`, `relationship`

To publish

- Use `Relationship` primarily for relationships between people.
- The `relationship` property is the type of relationship.
- Use the Relationship Vocabulary for the relationship type.
- The `subject` property is the originating member of the relationship.
- The `object` property is the receiving member of the relationship.

To consume

Don't assume that relationships are symmetrical.

See also

- Relationship Vocabulary
- `Follow`, for the more specific subscribe/publish model of social relationships
- `Offer`, for offering a relationship
- `Accept`, for accepting a relationship
- `Reject`, for rejecting a relationship

Remove

Extends `Activity`

Group Collection Management, ActivityPub, Activity

Importance High

Notes The `Remove` type is used to remove an object from a container. The `object` is the object to be removed; the `target` is the container to remove it from.

The most typical use is for a `target` that is a `Collection`. Other types that an actor could remove things from include a `Group` or `Organization`.

The Activity Vocabulary and ActivityPub specs differ on which property to use for the container. I say to follow the ActivityPub preference and use `target`.

Properties `object`, `target`, `origin`

To publish

- Use `object` for the object being removed.
- Use `target` for the container.

To consume Some publishers may use `origin` for the container object; check it if `target` is not provided.

See also

- `Add`, for adding an object to a collection
- `Move`, for moving an object from one container to another
- `Delete`, for deleting an object completely

Service

Extends `Object`

Group Actor

Importance Medium

Notes	In the social-network world, the `Service` type represents a server—either an ActivityPub server or another type. Note that a `Service` is different from an `Application`, which is better used for client software. Other types of service could be represented by `Service`. They should probably be further specified with multityping.
Properties	`name`, `summary`, `icon`, `url`
To publish	• Use `name` for the name of the service. • Use `summary` for a brief description of the service. • Use `icon` for a square, small image representing the service. • Use `url` for the web page for the service.
To consume	A `Service` can be an ActivityPub actor; if so, it will have `inbox`, `outbox`, `followers`, `following`, and `liked` properties.
See also	`Application`, for client software

TentativeAccept

Extends	`Accept`
Group	Event Management, Activity
Importance	Low
Notes	The `TentativeAccept` type is used to accept something with reservations. It is primarily useful for event management, to show that an acceptance is predicated on other factors (such as clearing other schedule conflicts). The condition(s) on which acceptance depends can be specified in the `context` property. You can use this type in other ways, similarly to using `Accept`, although most consumers would not expect it and may not process it correctly. For accepting an object into a container like a `Group`, `Collection`, or `Organization`, you can use the `target` property to identify the container. Event management isn't broadly implemented on the fediverse, and since this isn't well-defined for other uses, it's of low importance overall.
Properties	`object`, `target`

To publish
- Avoid using this type when `Accept` would work instead.
- Use `object` for the object being accepted tentatively, like an `Invite` activity.
- Use `target` if the object is being accepted into a container.

To consume If found, this type can safely be treated like an `Accept` activity.

See also
- `Accept`, for a more common acceptance
- `TentativeReject`, for tentatively rejecting an object

TentativeReject

Extends `Reject`

Group Event Management, Activity

Importance Low

Notes The `TentativeReject` type is used for rejecting something with reservations.

It is primarily useful for event management, to show that an invitation has been rejected, but might be changed based on other factors.

The condition(s) on which the decision depends can be specified in the `context` property.

You can use this type when `Reject` would be used, such as for rejecting a `Follow` activity. It's not well supported by most consumers, though, so avoid it when possible.

For rejecting an object submitted for inclusion into a container like a `Group`, `Collection`, or `Organization`, use the `target` property to identify the container.

Event management isn't broadly implemented on the fediverse, and since this isn't well-defined for other uses, it's of low importance overall.

Properties `object`, `target`

To publish
- Avoid this type when `Reject` could be used.
- Use `object` for the object being rejected.
- Use `target` if the object is being rejected from inclusion in a container.

To consume If found, this type can safely be treated like a `Reject` activity.

See also
- `Reject`, for the more conclusive activity
- `TentativeAccept`, for tentative acceptance of an object

Tombstone

Extends	`Object`
Group	Content Object, CRUD Activity
Importance	High
Notes	The `Tombstone` type represents an object that has been deleted. It's used because many links to the object might still remain out in the world, from before it was deleted. Finding and removing them all would be difficult, so instead the canonical representation of the object is replaced with a `Tombstone`.
Properties	`formerType`, `deleted`
To publish	• Use `formerType` for the former type of the object. • Use `deleted` for the datetime when the object was deleted. • Don't include any additional information about the object; the author deleted it for a reason, after all.
To consume	When it's found in a `Collection`, you can represent a `Tombstone` with "[deleted]" or skip it entirely
See also	`Delete`

Travel

Extends	`IntransitiveActivity`
Group	Geosocial, Activity
Importance	Low
Notes	The `Travel` type represents traveling from an origin, `origin`, to a destination, `target`. The `origin` and `target` properties should be `Place` objects. Unlike `Arrive` and `Leave`, `Travel` extends over a span of time, so properties like `duration`, `startTime`, and `endTime` are meaningful. Geosocial types aren't widely implemented, so the importance of this type is low.
Properties	`origin`, `target`, `duration`, `startTime`, `endTime`

To publish
- Use `origin` for the place of departure.
- Use `target` for the place of arrival.
- Use `duration`, `startTime`, and/or `endTime` to represent the time period.
- Carefully consider the actor's privacy when publishing a `Travel` activity, since it may include their current physical location.

To consume N/A

See also
- `Leave`, for the departure part of traveling
- `Arrive`, for the arrival part of traveling
- `Place`

Undo

Extends `Activity`

Group Meta-activities, ActivityPub, Activity

Importance High

Notes The `Undo` type represents undoing an activity. The `object` property is the activity being undone.

`Undo` is well-defined in ActivityPub for the `Like`, `Block`, and `Follow` activities. Many other activity types can be represented with an `Undo`.

Some types have more direct opposite activities, which should be used instead of `Undo`. For example, instead of undoing a `Create`, use `Delete`; instead of undoing `Add`, use `Remove`; instead of undoing `Join`, use `Leave`.

`Undo` makes most sense for activities that have side effects. It doesn't have a well-defined meaning for objects that aren't subtypes of `Activity`.

Undoing an activity that has previously been undone has no effect.

Properties `object`

To publish Use `object` for the activity to be undone.

To consume Some implementations may not provide an `id` property for the activity that is being undone, because it's not easy to retrieve. It may be possible to uniquely identify an activity from its `type`, `actor`, or `object` (for example, a `Follow` activity). In these cases, it is reasonable to proceed with undoing the activity.

See also N/A

Update

Extends	`Activity`
Group	CRUD Activity, ActivityPub, Activity
Importance	High
Notes	The `Update` activity type is for updating the properties of an object represented by the `object` property. The most common types of objects that can be updated are content objects, like `Note` or `Image`. Actor types (like `Person`) and `Question` activity types can also be updated. Other activity types may be updateable, especially for descriptive properties like `summary` or `icon`. However, changing more semantic properties might not have the desired effect. For example, changing the `object` of a `Follow` activity to follow a different `Person` might not initiate the full series of steps necessary to start that `Follow` process. Changing an object's `id` property is not possible. Some other properties, especially collections that are managed by the server, might not be possible to change. When posted to an actor's `outbox` as part of the ActivityPub API, the `object` can include only the properties being changed. However, in other contexts, such as when being delivered with the ActivityPub protocol or retrieved with a GET, the `object` should include the full, updated set of properties.
Properties	`object`
To publish	• When posting to the ActivityPub API, include only the properties that have changed. • In other contexts, include the full `object` with all properties, including the ones that have changed.
To consume	Receiving an `Update` is an opportunity to either flush the cache for the `object` or replace it with the properties in `object`.
See also	• `Create`, for creating an object • `Delete`, for deleting an object • `Undo`, for undoing an activity (which can then be redone with different properties)

Video

Extends	`Document`
Group	Content Object, Object
Importance	High
Notes	The `Video` type represents a digital video. The `id` is the URL of the JSON-LD metadata about the video; the `url` is the URL for the binary format of the video. The `mediaType` property is the content type of the video, typically a web-friendly format like WebM or MP4. For services that embed a video into a web page, like YouTube, it's difficult to get a URL that leads to the web-friendly binary data for the video. This may be better represented as a `Page`. There's no agreed-upon ID namespace for video content that isn't easily linked to, like a film or a TV episode. Two `Video` objects with different `id` values may represent the same real-world film or episode. A `Video` object can go through a full create-update-delete cycle, like other content types.
Properties	`url`, `mediaType`, `name`, `summary`, `image`, `icon`, `duration`
To publish	• Use `url` for the URL of the binary representation of the video. • Use `mediaType` to indicate the content type. • The `name` property gives the title of the video, if any. • The `summary` property is a brief description of the video. • The `image` property can be used to provide a still from the video. • The `icon` property is for a square representation of the video. • The `duration` property can give the duration of the video.
To consume	If `mediaType` is not provided, it may be guessed from the file extension, if any, at the end of the `url` property.
See also	• `Page`, for a page embedding a video, like a YouTube video • `View`, for the activity of watching a video

View

Extends	`Activity`
Group	Content Experience, Activity
Importance	Medium
Notes	The `View` type is used when the actor has experienced a visual content object, like an `Image` or a `Video`. The `object` can also be a `Link`, if the thing being watched does not have an AS2 representation. This activity typically has an extent in time, so `duration`, `startTime`, and `endTime` are all relevant. Social experience-sharing activities are great for social networks, but this type is not well implemented yet on the fediverse.
Properties	`object`, `result`
To publish	• Use `object` for the object watched. • Use `duration`, `startTime`, and/or `endTime` to show how long the viewing took and when it started and ended. • Use `result` for any result of viewing the video or image object, such as notes or a review.
To consume	Be prepared for `Document` subtypes like `Image` and `Video` as well as `Link`.
See also	• `Video` • `Image` • `Read` • `Listen`

APPENDIX B
Activity Streams 2.0 Properties

This appendix describes each of the important properties defined in AS2. I cover what type of object they can apply to, what values they can have, what they're used for, and what to watch out for. I also give tips for publishers and consumers to maximize interoperability.

For each property, I include the following information:

Information	Explanation
Type	The type of object this property applies to
Value	The types and typical values of this property, including whether plural values are allowed
Group	A rough category of related properties, created by me
Importance	A subjective evaluation of the importance of this property
Notes	My notes on the property
To publish	Tips on publishing objects with this property
To consume	Tips on consuming objects with this property
See also	Related properties that may be worth considering

Most of the value types are typical of programming language types: strings, integers, floating-point numbers, Booleans. Many are URI or URL types. Most important, the `id` type can be an object, a URL, or an array of objects and/or URLs. Handling `id`s or JSON objects is discussed pretty thoroughly in Chapter 2, including how to handle each of the value types.

I wish I could say that I love all the AS2 properties equally, but they aren't my kids, so I don't have to do that. Some of these properties are under-defined and not easy to use. In the notes and tips, I try to err on the side of interoperability and rich, interesting ActivityPub applications, and if you have other priorities, well, agree to disagree.

Groups

I organize the AS2 properties into groups, so I want to give an idea of what those groups actually mean. Note that this name is unrelated to the `Group` type!

Group	Properties
Activity	Properties important to the structure and functioning of activity objects
Actor	Properties usually defined only for an ActivityPub actor
Addressing	Properties for the addressing and delivery of activities and content
Collections	Properties of collections and collection pages
Content	Properties for content objects, like text, images, audio, and video
Context	Properties that define the situation an activity was done for
Fundamental	Properties necessary for defining objects in AS2
Geographical	Properties that define geographical extent
Image	Properties that give two-dimensional visual representation of an object
Link	Properties applicable to only the `Link` type
Profile	Properties applicable to only the `Profile` type
Question	Properties used in polling or open-ended questions
Reactions	Properties for reactions to an object
Relationship	Properties that define relationships between people or things
Time	Properties used for defining extent in time
Tombstone	Properties of deleted objects

These are informational only; they don't have any functional impact. I find that they help me when I'm looking at a property to determine whether it's necessary for what I'm working on or whether I can ignore it for now.

accuracy

Type	`Place`
Value	Floating-point number, like `94.5`, representing a percentage
Group	Geographical
Importance	Low
Notes	It's somewhat embarrassing to start off with one of the more difficult properties used in ActivityPub. This property is supposed to indicate the accuracy of a location and derives from the accuracy property used in the web Geolocation API. However, it's expressed as a percentage, like 92.5. It's not clear what this is a percentage of, so it's hard to use. This should probably be used only when a location is obtained from the Geolocation API itself.

To publish	Avoid using this property, as it's not well-defined and can be misinterpreted.
To consume	Ignore this property, as it's not well-defined and can be misinterpreted.
See also	• `radius` • `latitude` • `longitude` • `altitude`

actor

Type	`Activity`
Value	`id`, usually but not always singular
Group	Activity
Importance	High
Notes	This property represents the actor who performed the activity. It's used for attributing the responsibility for the activity, for authorization on making changes to the activity or its results.
To publish	• Include this property on most activities, unless it is unneeded from context (for example, in the `outbox` of an actor, or when creating an activity). • Include a functional representation of the actor, since it is often used in displaying the activity.
To consume	• Be prepared for plural values of `actor`, even if they are unusual. • `actor` can also be a `Link`.
See also	`attributedTo`

alsoKnownAs

Type	`Object`

Value	URI, often plural. Note that the URI doesn't have to be a URL. For example:
	`"alsoKnownAs": [`
	`  "acct:username@domain.example",`
	`  "did:example:123456789abcdefghi",`
	`  "tag:example.com,2024:an-example"`
	`]`

Group	Fundamental
Importance	Medium
Notes	This property lists other identifiers that refer to the same real-world object. In ActivityPub, this property is used for data portability, to show that two ActivityPub accounts are used for the same person (see Chapter 5). It can also be used to connect an ActivityPub account with another identity system, like WebFinger.
To publish	Use the `id` representation (just a URL) for each object, even for ActivityPub actors.
To consume	• Be prepared for a single object or `id`—usually because of JSON-LD compaction. • This property is used to connect ActivityPub actors with other identity systems, so many of the URLs in this array will not use the HTTPS protocol. • The other identity should also have a link back to this object, in whatever mechanism it uses. For ActivityPub, that means an `alsoKnownAs` that includes this object's `id`.
See also	`id`

altitude

Type	`Place`
Value	Floating-point number, like `200.0`, representing the place's altitude above sea level in meters (or in the units defined in `units`)
Group	Geographical

Importance	Medium
Notes	Defining altitude for places is not too bad in AS2. One big issue is that there is only one `units` property per place, so you have to use the same units for `radius` and `altitude`.
To publish	Provide altitude in meters if possible, to save your consumer a lot of conversion math.
To consume	Be prepared for altitude values in units other than meters.
See also	• `units` • `radius`

anyOf

Type	`Question`

Value	`id`, usually an array of available responses, like this:
	`[`
	`  {`
	`  "id": "https://social.example/note/1",`
	`  "type": "Note",`
	`  "name": "Chips"`
	`  },`
	`  {`
	`  "id": "https://social.example/note/2",`
	`  "type": "Note",`
	`  "name": "Ice Cream"`
	`  }`
	`]`

Group	Question
Importance	Medium
Notes	This property allows multiple-answer polls and questions. Responders can reply with more than one of the defined answers.
To publish	For maximum interoperability, use an array with `Note` objects, each of which has a unique `name` property.
To consume	• Watch for single values, which can happen if the poll is compacted. • Watch for URL values.
See also	`oneOf`

attachment

Type `Object`, usually but not always text types like `Note` or `Article`

Value `id`, of an `Object` or `Link`, often but not always plural. For example:

```
"attachment": [
   {
      "id": "https://social.example/image/17",
      "type": "Image",
      "summary": "Me and Michele on the beach",
      "url": "https://social.example/uploads/beach.jpg"
   },
   {
      "id": "https://social.example/image/19",
      "type": "Image",
      "summary": "Later, at supper on the wharf",
      "url": "https://social.example/uploads/supper.jpg"
   }
]
```

This property can also be `Link` objects:

```
"attachment": [
  {
     "type": "Link",
     "name": "My Blog",
     "href": "https://evanp.me"
  }
]
```

Group Content

Importance High

Notes Similar to an email attachment, this is for secondary content that supports the main content object. Note that the actual content is not "attached," or included in the body of the object, but rather linked by its `url` or `href`. Multiple objects can be in the attachment array, or just a single one.

To publish

- Use this property primarily for content types like `Document`, `Image`, `Video`, and `Audio`.
- This property is also useful for attaching detailed information about a URL, in the form of a `Link`, used in the main content of the object.

To consume	• This property can have a single value because of JSON-LD compaction. • This property may include `id` URLs, not objects. • `attachment` may include noncontent `Object` types. • `attachment` may contain one or more `Link` objects.
See also	`tag`

attributedTo

Type	`Object`, `Link`
Value	`id`, usually but not always singular
Group	Content
Importance	High
Notes	This property represents the creator, author, or owner of the object. The value is usually but not always an actor type like `Person`, `Organization`, or `Application`.
To publish	• Include this property on most objects. • Try to include a brief representation of the actor, since it is often used in displaying the object.
To consume	• Be prepared for plural values of `attributedTo`, even if they are unusual. • `attributedTo` can also be a `Link`.
See also	`actor`

audience

Type	`Object`
Value	`id`, often plural
Group	Addressing
Importance	Low
Notes	I'm not a fan of this property; I think it's kind of a mess. It's not considered an addressing property in AS2, but it is in the ActivityPub spec. I think the intent was to specify a way that an object could be visible to an audience, but not delivered to that audience, and instead it got lumped into the delivery properties anyway.

To publish	Avoid this property. Use `to` or `cc` instead.
To consume	Treat this property as synonymous with `to` or maybe `cc`.
See also	• `to` • `cc`

bcc

Type	`Object`
Value	`id`, often plural
Group	Addressing
Importance	Medium
Notes	I understand the intent of this property, but along with `bto`, it really complicates the implementations of ActivityPub. It stands for *blind carbon copy*, which combines an ableist description with some archaic technology to make for an almost incomprehensible property. You can use this property to deliver an activity or object to an actor without anyone else knowing, with the downside that the reason it was delivered is opaque to everyone, including the recipient. This makes it hard for other processors to determine whether an actor has read access to the object.
To publish	• Avoid this property. Use `cc` instead. • If this property is included in an object, strip it from the object before delivering it over the REST API or via ActivityPub protocol inbox delivery.
To consume	• Treat this property as synonymous with `cc`. • This property, which is usually an array, may be a single object because of compaction. • No behavior is defined for delivery to a `Link` object, but that pattern might be useful for delivering through a bridge to another protocol.
See also	• `bto` • `cc`

bto

Type	`Object`
Value	`id`, often plural
Group	Addressing
Importance	Medium
Notes	Similar to `bcc`, but more confusing since it's not parallel to any email headers. Used for secret delivery to a primary audience. ActivityPub doesn't make much of a distinction between primary and secondary audiences, though, so this makes it almost identical to `bcc`. It's hard for other processors to tell why an actor might receive or have access to an object, because this property is stripped by publishers before delivery.
To publish	• Avoid this property. Use `to` instead. • If this property is included in an object, strip it from the object before delivering it over the REST API or via ActivityPub protocol inbox delivery.
To consume	• Treat this property as synonymous with `to`. • This property, which is usually an array, may be a single object because of compaction. • There isn't a defined behavior for delivery to a `Link` object, but it might be useful for delivering through a bridge to another protocol.
See also	• `to` • `bcc`

cc

Type	`Object`
Value	`id`, often plural
Group	Addressing
Importance	High

Notes	This is an important addressing property. It's used to indicate a secondary audience for an object or activity, such as with email. Most ActivityPub software doesn't seem to use this much differently than `to`, however. You can use it to separate stuff that the author felt was important to call attention to, versus those objects that were shared secondarily, which could maybe go in a different stream.
To publish	Use a brief representation for objects to give some extra metadata to consumers.
To consume	• Treat this property as synonymous with `to`. • This property, which is usually an array, may be a single object because of compaction. • No behavior is defined for delivery to a `Link` object, but it might be useful for delivering through a bridge to another protocol.
See also	• `to` • `bcc`

closed

Type	`Question`
Value	Many possibilities! • A Boolean, like `true` or `false` (is this closed or not) • A datetime like `2024-05-25T00:30:00Z` (when the question was closed) • An `Object` or `Link` (the results of the poll)
Group	Question
Importance	Medium
Notes	This is probably the most chaotic property in AS2, but I have a certain fondness for it. Its meaning varies a lot depending on the type of its value.
To publish	• Use the datetime format. • Use `results` to show the results of the question.
To consume	• Be prepared for all four types in the value. • If the value is a Boolean or datetime, look for the results in the `results` property instead.
See also	• `results` • `startTime` • `endTime`

content

Type	`Object`
Value	The content of the object; typically HTML5 text
Group	Content
Importance	High
Notes	For document-like objects, this is the content of the object. It's primarily used for text content in `Note` and `Article` objects, so it is almost always HTML5. The data format of this property can be different based on the `mediaType` of the `Object`. For example, if `mediaType` is `text/x-markdown`, the contents can be Markdown text. A little further afield, the `mediaType` on an `Image`, for example, could even be a binary file type, like `image/gif`. In that case, the content property would need to be a JSON-compatible text encoding of the binary file, using something like base64. Unfortunately, there's no AS2 property to say which encoding to use, so it's difficult for consumers to figure out what to do with all that text (except make an educated guess). This property is mostly experimental, because the vast preponderance of `content` properties on the social web are HTML5, to the point where most consumers will not check the `mediaType` property and will just treat the string as HTML. Especially for `Note` and `Article` objects, there's not much point including anything but HTML in `content`.
To publish	• Use this field for text content of text-like object types. • Use sanitized HTML.
To consume	Sanitize the HTML in this property.
See also	• `source` • `summary` • `url` • `mediaType`

contentMap

Type `Object`

Value A JSON object mapping language tags as defined in RFC 5646 to the content of the object, such as this:

```
"contentMap": {
  "en": "Hello, World!",
  "fr": "Bonjour, le Monde !"
}
```

Group Content

Importance High

Notes For document-like objects, this is the content of the object. Note that its interpretation can change based on the `mediaType` property. By default, the values can contain HTML5 content.

To publish

- Use this field for text content of text-like object types.
- Include translations only if the author has provided them; automatic translations are better handled on the consumer side.
- A `contentMap` with a single language tag is a good way to indicate the language used in the content.
- Use sanitized HTML.

To consume

- Choose the right language tag based on the user's language preferences. RFC 4647 outlines some tools and techniques for matching languages.
- Sanitize the HTML in this property.

See also

- `content`
- `source`
- `summary`
- `url`
- `mediaType`

context

Type	`Object`
Value	`id`, usually singular
Group	Context
Importance	Medium
Notes	This is an intentionally vague property and could represent any context for any type of object—for example, the mood a person was in when they did a `Travel` activity. However, the main use of `context` on the fediverse today is to represent the conversation that a content object belongs to. A conversation, for this discussion, includes an original post, all its replies, all replies to those replies, and so on. For example, if a `Note` object has a `context` value that refers to a `Collection`, that collection may include the full tree of notes that make up the conversation, linked by the `inReplyTo` property.
To publish	• Use this field to refer to a `Collection` with the conversation tree for the current object. • Use more-specific properties, like `location` or `instrument`, to refer to more-specific contexts. • An extension property (see Chapter 5) is a better use.
To consume	• If the value of this property is a `Collection`, and the object has an `inReplyTo` property, the value probably represents the conversation tree. • Some publishers use URL formats for `context` that aren't dereferenceable like `tag:` URIs. In this case, you can consider the object "in the same conversation" as other objects with identical `context` values. • Be prepared for values of `context` that aren't collections and don't represent the conversation. As I've mentioned, the property is intentionally vague!
See also	• `instrument` • `location`

current

Type	`Collection`
Value	`id`, singular, usually a `CollectionPage` but can be a `Link`
Group	Collections
Importance	Low
Notes	In a paged collection, this represents the page that has the most recent items in it. For a reverse chronologically ordered collection, it will usually be identical with the `first` property. For a chronologically ordered collection, it will often be identical with the `last` property.
To publish	Use this property to indicate the `CollectionPage` or `OrderedCollectionPage` with the most recent items in it.
To consume	Be prepared for a `Link` value.
See also	• `first` • `last`

deleted

Type	`Tombstone`
Value	`datetime`
Group	Tombstone
Importance	Medium
Notes	For a deleted object, this represents the date it was deleted.
To publish	Use this for `Tombstone` objects to represent their deletion date.
To consume	N/A
See also	• `published` • `updated` • `closed`

describes

Type	`Profile`
Value	`id`, usually singular
Group	Profile
Importance	Low
Notes	For a `Profile` object, which represents a specific kind of web page, this property links to the object the profile describes. This is primarily useful for profile pages, in which case the `describes` value would usually be a `Person` or `Organization`.
To publish	• Use this for `Profile` objects, to represent the person or organization it is the profile for. • Don't use it for any other kind of object, even though it seems pretty useful.
To consume	N/A
See also	N/A

duration

Type	`Object`
Value	A formatted string, as defined in the XML Schema data types specification, and typically consisting of a prefix and then a number followed by a unit. Here are examples: • `P2Y`, for 2 years • `P3M`, for 3 months • `P12D`, for 12 days • `PT8H`, for 8 hours • `PT3M`, for 3 minutes (note the difference from months!) • `PT15S`, for 15 seconds • `P1DT12H`, for 1 day plus 12 hours
Group	Time
Importance	Low
Notes	For objects that have spatial extent, like a `Video` or an `Event`, this property represents how long the object will or did last. The duration format borrowed from XML Schema is somewhat opaque.

To publish	• If possible, prefer `startTime` and `endTime` instead of using this property. • Use seconds as the units if at all possible, to reduce parsing overhead for consumers.
To consume	Consider using an open source library for parsing these duration values.
See also	• `startTime` • `endTime`

endpoints

Type	`Object`
Value	A JSON object with zero or more of these properties, each of which should be a URL: • `proxyUrl` • `oauthAuthorizationEndpoint` • `oauthTokenEndpoint` • `provideClientKey` • `signClientKey` • `sharedInbox` This property can also be a URL to an object with these properties.
Group	Actor
Importance	High
Notes	This property is a grab bag of useful URLs related to an actor. I talked about `proxyUrl` in Chapter 3 and `sharedInbox` in Chapter 4. The OAuth-related properties can be used for discovering the necessary endpoints for the OAuth authorization flow. The client key properties are kind of mysterious; they describe a form of cryptographic client authentication that I haven't seen in use.
To publish	Use the inline form of `endpoints`, not the remote URL form.
To consume	Be aware that this property can also have a URL as a value.
See also	`streams`

endTime

Type	`Object`
Value	`datetime`
Group	Time
Importance	Medium
Notes	For objects with an extent in time, this is the end of that time extent. This is primarily useful for `Event` but may be useful for content that is available for only a limited time or for a `Question`.
To publish	Use in conjunction with `startTime`.
To consume	N/A
See also	`startTime`

first

Type	`Collection`
Value	`id` of a `CollectionPage` or `Link`, singular
Group	Collections
Importance	High
Notes	For paged collections, this is the first page in the collection. A typical strategy for looping through all the pages of the collection is to start with `first` and then follow the `next` property of each page until there is no `next`. Note that in reverse chronologically ordered collections, the `first` property will refer to the page with the most recently added items, not the oldest items. This can be really tricky to remember!
To publish	• For any paged collection, include this property. • It can be useful to include the properties of the `first` collection page, including the `items` property, to give a peek at the contents of the collection.
To consume	Check for the object form of this property; it may include useful information, like `items`, that will save you additional requests.
See also	• `last` • `current` • `next` • `prev`

following

Type	`Object`
Value	`id` of a `Collection` or `OrderedCollection`, singular
Group	Actor
Importance	High
Notes	This is a collection of all the other actors that the actor has followed. If it is an `OrderedCollection`, it has to be in reverse-chronological order, usually by the time that the `Follow` activity was accepted.
To publish	• Use an `OrderedCollection`. • Some ActivityPub consumers won't correctly handle this property unless it is in its `id` representation (just a URL). For maximum interoperability, use the `id` representation.
To consume	Watch for the object form of this property; it may save you additional requests.
See also	`followers`

followers

Type	`Object`
Value	`id` of a `Collection` or `OrderedCollection`, singular
Group	Actor
Importance	High
Notes	This is a collection of all the other actors that follow this actor. I have worked on this standard for more than a decade, and I still mix up `followers` and `following` on a regular basis. The best mnemonic I've been able to come up with is that the collection of people that `I` follow has an `i` in the name. If `followers` is an `OrderedCollection`, it must be in reverse-chronological order, usually in the order that the actor accepted the follow requests.

To publish	• Use an `OrderedCollection`. • Some ActivityPub consumers won't correctly handle this property unless it is in its `id` representation (just a URL). For maximum interoperability, use the `id` representation.
To consume	Watch for the object form of this property; it may save you additional requests.
See also	`following`

formerType

Type	`Tombstone`
Value	`object` type, usually singular
Group	Tombstone
Importance	Medium
Notes	This indicates the type of this object before it was deleted. This property can be useful for indicating that the object is a deleted `Image` rather than a deleted `Article`, for example.
To publish	Include this property for `Tombstone` objects to save some typing information.
To consume	• Watch for the plural form of this property, as with `type`. • If this property is missing, assume that the former type is `Object` for any further processing.
See also	`type`

generator

Type	`Object`

Value	`id`, usually singular, such as the following:
	`"generator": {`
	`  "id": "https://drawing.example/id",`
	`  "type": "Application",`
	`  "icon": "https://drawing.example/icon",`
	`  "name": "Drawing application",`
	`  "url": "https://drawing.example/"`
	`}`

Group	Context
Importance	Medium
Notes	This property is for identifying the application used for creating an object or activity. It is distinct from `instrument` in that it is primarily related to creation of objects.
To publish	This property is the best way to indicate the client application that created the object.
To consume	• Watch for the plural form of this property. • Linking to the `url` of the generator can help other actors find the software so they can use it too.
See also	`instrument`

height

Type	`Link` (not `Object`!)

Value	Nonnegative integer, as in the following:
	`"url": {`
	`  "type": "Link",`
	`  "width": 100,`
	`  "height": 100,`
	`  "mediaType": "image/png",`
	`  "href": "https://domain.example/image.png"`
	`}`

Group	Link
Importance	Medium
Notes	Frustratingly, this property is only for `Link` objects. The best way to use it for an `Image` or `Video` is to have a `Link` object for the `url` property, and include the height (and width) there.
To publish	• Only for `Link` objects. • Including multiple `url` properties with different heights and widths can be helpful for clients picking an appropriately sized image.
To consume	If a `Link` is in an array, you can sometimes pick the best size image or video by comparing `height` and `width`.
See also	`width`

href

Type	`Link`

Value	URL, as in the following:
	`"icon": {`
	`  "type": "Link",`
	`  "href": "https://domain.example/image.png"`
	`}`

Group	Link
Importance	High
Notes	This property is the whole point of a `Link` object.
To publish	• Most ActivityPub processors, as of this writing, will expect an `https:` URL for the `href` value. • If you want to use another protocol, like IPFS, you should include multiple `Link` values in an array, to allow consumers to pick one with a protocol they can use (HTTPS is the fallback case).
To consume	If a `Link` has an `href` with a protocol you don't recognize, check whether other links in the same array have an HTTPS protocol instead.
See also	`url`

hreflang

Type	`Link`
Value	A language tag as defined in RFC 5646, such as the following: • `en` (English) • `fr-CA` (Canadian French) • `abe` (Western Abenaki)
Group	Link
Importance	Medium
Notes	This property gives the language used in the resource referenced in the link. It's especially useful for written content, for videos with humans speaking, or for spoken or sung audio. Determining what "the same" language is can be tricky; RFC 4647 outlines some tools and techniques for matching languages.
To publish	• If available, include the `hreflang` value in your `Link` objects. • If there are multiple language versions of the same resource, use an array of `Link` objects, each with a unique `hreflang` value.
To consume	• If a `Link` has an `hreflang` with a language code your user doesn't understand, check whether other links in the same array have an `hreflang` with a language code they do understand. • Consider using a library for parsing and comparing language tags, as the variants can get really complex.
See also	`contentMap`

icon

Type `Object`

Value `id`, often plural, often `Link` objects, as follows:

```
"icon": [
  {
     "type": "Link",
     "width": 128,
     "height": 128,
     "name": "Calendar Bot",
     "mediaType": "image/png",
     "href": "https://social.example/files/smallicon.png"
  },
  {
     "type": "Link",
     "width": 512,
     "height": 512,
     "name": "Calendar Bot",
     "mediaType": "image/png",
     "href": "https://social.example/files/largeicon.png"
  }
]
```

The value can also be one or more `Image` objects, such as the following:

```
"icon": {
     "type": "Image",
     "name": "Calendar Bot",
     "url": {
       "type": "Link",
       "width": 128,
       "height": 128,
       "mediaType": "image/png",
       "href": "https://social.example/files/smallicon.png"
     }
}
```

Group	Image
Importance	High
Notes	This property is an important part of representing actors, objects, and activities on the social web. Unlike `name`, it is not strictly required, but I really think even the briefest object representations should include it. The icon should be square and "small," which as of this writing means somewhere in the low hundreds of pixels in width and height, and in the hundreds of kilobytes in file size. This property is distinct from `image` in that it should be small and square; `image` can be larger and have a different aspect ratio.
To publish	• Include an `icon` with most object types. • Use the `Link` type for icons, including metadata about size and file type. • When possible, include multiple sizes for the same icon, so that consumers can pick the one closest to their display size without too much scaling. • Consider using a content-delivery network (CDN) for icon files, since they are loaded fairly often. • If using multiple values, avoid mixing different types (`Link` and `Image`).
To consume	• Be prepared for both `Link` and `Image` format. • If this property has an array value, pick the item that matches your display requirements.
See also	`image`

id

Type	`Object`, `Link`
Value	URL, singular
Group	Fundamental
Importance	High
Notes	This is the globally unique identifier for any object in the ActivityPub universe. This property doubles as a location on the web that can be used to fetch all available information about that object. So, if an activity includes a brief representation of the `actor` property value, a consumer can use the `id` property to get a full representation. The ActivityPub specification requires objects to have an `id` property, unless the object is "intentionally transient." This is ambiguous, and I don't recommend depending on it. Providing identities for most object types is easy and it enables consumers to confirm the properties of objects they receive and to share them by `id`. Providing `id` values is an important part of being a good participant in the social web. Theoretically, you can include other dereferenceable URL types in the `id` property (that is, URLs that can be used to get the content of resources). However, as of this writing, no protocols are used for ActivityPub `id` URLs except `https`. It is theoretically permissible to use URL fragments (the part after the `#`) in ActivityPub identifiers. However, there's no well-defined way to dereference a fragment in a JSON-LD document. This makes it hard for consumers to dereference your identifiers, so avoid it if you can. An `id` on a `Link` can give a unique identifier to the relationship between the containing object and the resource at the `href` URL. However, in practice it's usually much less important than including `id` values for `Object` and its subtypes.

To publish

- Include a unique `id` for every object you publish over the ActivityPub API or protocol.
- Use only `https` for your `id` property.
- Use persistent and unique `id` URLs.
- Don't use URL fragments in `id` properties.
- Avoid using query parameters (the part after the `?`), since some consumers may not handle them correctly.
- Don't change the `id` paths. If you restructure the routes in your ActivityPub software, keep the old routes for backward compatibility, so old `id` values will still be dereferenceable.
- `id` values should be distinguishable with simple string comparisons. Don't depend on consumers to do complicated comparisons between equivalent URLs. Always use lowercase domain names, and always use query parameters in the same order (if at all).
- Don't include properties that users can change in the path of your `id` URLs. In particular, if users can change their username, don't include it in the `id` for that actor or any of the activities or content they create.

To consume

- If an `id` value is missing or uses a protocol that your software doesn't know how to dereference, treat it gingerly. You won't be able to reuse or do any confirmation on the object.
- Some `id` values will include query parameters (like `?foo=bar&baz=quux`).
- Some `id` values will include URL fragments (like `#publicKey`). As a best guess, these can be dereferenced by retrieving the full document and then extracting the top-level property of that type. Some URL fragments may not be dereferenceable, though.
- Compare `id` values stringwise.

See also `url`

image

Type	`Object`
Value	`id`, often plural, usually a `Link` but can also be an `Image`
Group	Image
Importance	Medium
Notes	Similar to `icon`, this is an image of the object. Unlike an icon, it can be large and doesn't have to be square. For ActivityPub actors, this property is often used to provide a background image on the actor's profile page.
To publish	• Use the `Link` type for images, including metadata about size and file type. • When possible, include multiple sizes for the same image, so that consumers can pick the one closest to their display size without too much scaling. • Consider using a content-delivery network (CDN) for image files, since they are loaded fairly often. • If using multiple values, avoid mixing different types (`Link` and `Image`).
To consume	• Be prepared for both `Link` and `Image` format. • If this property has an array value, pick the item that matches your display requirements.
See also	`preview`

inbox

Type	`Object`
Value	`id`, usually a URL, usually singular
Group	Actor
Importance	High
Notes	This is the collection of all activities an actor has received and/or created. The actor can fetch the collection to read their incoming activities, as described in Chapter 3; remote actors can post to the endpoint to deliver new activities, as described in Chapter 4.

To publish
- Some ActivityPub protocol implementers won't handle the object form of this property, so always use the `id` representation.
- Some ActivityPub implementers use this property to duck-type actors (that is, an ActivityPub object is treated as an actor if and only if it has an `inbox` property).

To consume
- Be prepared for both URL and object representations.
- The ActivityPub specification doesn't specifically state that this property should be singular. Most current implementations use a single URL, but having multiple URLs in this property is theoretically possible. In that case, you may need to choose a URL by arbitrary criteria—like the first one in the array.

See also `outbox`

inReplyTo

Type `Object`

Value `id`, usually an `Object`, usually singular, as in the following:

```
"inReplyTo": "https://magazine.example/2024/05/26/article1"
```

It can also be in object form:

```
"inReplyTo": {
  "id": "https://magazine.example/2024/05/26/article1",
  "type": "Article",
  "name": "An Interesting Article in a Magazine"
}
```

Group Content

Importance High

Notes This property forms the basis of conversational trees on the social web. Once a new object is created, other objects (often `Note` objects) are created "in reply to" that object, more or less like comments. Replies to those objects will form a tree structure.

To publish
- Use a brief representation if possible.
- Avoid using an array value here. In particular, it's not necessary to represent the entire conversational tree in this property, just the object that this object is immediately replying to.

To consume
- Be prepared for `id` representation or fuller object representations.
- Be prepared for a `Link` value.

See also

- `context`
- `replies`

instrument

Type `Activity`

Value `id`, usually an `Object`, can be plural. For example, the `instrument` property for a `Listen` activity might include both the streaming service and the music-playing app:

```
"instrument": [
  {
    "id": "https://streaming.example/",
    "type": "Service",
    "name": "Music Streaming Service",
    "url": "https://streaming.example/home"
  },
  {
    "id": "https://player.example/",
    "type": "Application",
    "name": "Music Player",
    "url": "https://player.example/download"
  }
]
```

Group Context

Importance Low

Notes This property defines tools, especially software, that were used for completing the activity. It is similar to `generator` but includes activity types that don't result in the creation of new objects. Ideally, this would be shown in user interfaces so that other users could find and use the `instrument`, but few social web apps do this as of this writing.

To publish

- Use a brief representation if possible.
- Include a `url` for linking in consumer software.

To consume

- Be prepared for plural values.
- Be prepared for a `Link` value.
- If possible, provide a link in your user interface to let users find and try the `instrument` software themselves.

See also `generator`

items

Type `Collection`

Value Array of IDs, or sometimes a single `Object` or `Link`. Here is an `items` property for the `replies` collection of an `Article`:

```
"items": [
  {
    "id": "https://social.example/note/1",
    "type": "Note",
    "actor": "https://social.example/user/13",
    "content": "Good blog post!"
  },
  {
    "id": "https://reader.example/comment/775",
    "type": "Note",
    "actor": "https://reader.example/user/eric",
    "content": "Bad blog post!"
  }
]
```

Group Collections

Importance High

Notes This is where the contents of a collection are stored. Because of JSON-LD compaction, the value may be a single `Object` or `Link`.

The recommended maximum number of items in this array depends on the application. For most ActivityPub software, having about 20 items in the `items` array is a reasonable heuristic.

To publish

- Use the more specific `orderedItems` property for `OrderedCollection` or `OrderedCollectionPage` objects.
- Use a brief representation for each item if possible.
- In a paged collection, don't include this property on the collection, only on the pages.
- Don't mix `id` representations (just a URL) and brief or fuller representations (JSON objects).
- Don't include an `items` and an `orderedItems` property in the same object.

To consume
- Be prepared for either URLs or JSON objects or both.
- Be prepared for singular values.
- Be prepared for `Link` values.
- If the object is an `OrderedCollection` or `OrderedCollectionPage`, treat this collection as ordered.

See also `orderedItems`

last

Type `Collection`

Value `id` of a `CollectionPage` or `Link`, singular

Group Collections

Importance Medium

Notes For paged collections, this is the last page in the collection. It's possible, but unusual, to start with the `last` page and scroll forward by following the `prev` property.

For reverse-chronologically sorted collections, the object that was added earliest to the collection will be the last item on the last page. This can be conceptually challenging; be careful with it!

To publish For any paged collection, include this property.

To consume Check for the object form of this property; it may include useful information, like `items`, that will save you additional requests.

See also
- `first`
- `current`
- `prev`

latitude

Type `Place`

Value Signed floating-point number, like `45.63691`

Group Geographical

Importance Medium

Notes	For a point-like place, this provides the latitude of the point. Note that it's the signed numeric representation, and not a string with *N* or *S* at the end. Note that the name is full, and not the abbreviation *lat*.
To publish	Publish this only in conjunction with a related `longitude` value.
To consume	N/A
See also	`longitude`

liked

Type	`Object`
Value	`id`, usually a URL, for an `OrderedCollection` or `Collection`
Group	Actor
Importance	Medium
Notes	This is a collection of all objects that the actor has liked. If it is an `OrderedCollection`, it is in reverse-chronological order. The items will usually be unique by object `id`; liking something twice will not add it multiple times to the `liked` collection. It is easily confused with the `likes` property, which includes all the `Like` activities for an object. `liked` contains the objects themselves, not the activities. The `liked` collection should usually be filtered to return only items that the client can read.
To publish	Some ActivityPub software may not recognize the object form of this property, so use the URL form.
To consume	Be prepared for an object in this property.
See also	`likes`

likes

Type `Object`

Value `id` of an `OrderedCollection` or `Collection`. Here's an example:

```
"likes": {
   "id": "https://social.example/note/335/likes",
   "type": "OrderedCollection",
   "summary": "Likes of a note",
   "totalItems": 224,
   "first": "https://social.example/note/335/likes/page/1"
}
```

Group Reactions

Importance Medium

Notes This is a collection of all the `Like` activities where the object was the `object` of the activity.

Usually, the `Like` activities will be unique by actor ID; liking something twice will not put multiple activities in the collection.

This property is easily confused with `liked`, which contains the objects an actor has liked.

The collection will sometimes be filtered, and not all activities will be visible.

It can be an `OrderedCollection` or a `Collection`. If it is ordered, it must be ordered reverse chronologically.

To publish

- Include a functional representation for this property where possible, including `totalItems`.
- Filter the contents according to the consumer's authorization. Don't include items that the consumer can't read.
- Include `Like` activities by URL.

To consume This property can be in either an `id` representation or an object representation.

See also `liked`

location

Type	`Object`

Value	`id`, usually singular:
	`"location": {`
	`  "id": "https://places.example/ca-mtr",`
	`  "type": "Place",`
	`  "name": "Montreal, Quebec, Canada"`
	`}`

Group	Geographical
Importance	Medium
Notes	This property is for physical locations, usually on planet Earth. It's not for the location of digital objects, like URLs or disk drives. For many objects, this is the physical location where the object is currently or habitually. For a person, this may be their current location or their hometown. For activities, this is the physical location where the activity took place. For digital content, this is usually the physical location where the content was created. For the location represented in the content, if different from where the content was created, it may be better to use the `tag` property. For example, a painting created in Berlin representing a sunset in Maui might have `location` equal to Berlin and a `tag` item for Maui.
To publish	• Use a `Place` object for this property, with a brief representation. • Avoid using the plural form of this property, as some consumers won't know how to handle it.
To consume	This property can be in either an `id` representation or an object representation.
See also	`tag`

longitude

Type	`Place`
Value	A singular signed floating-point number, like `-72.14989`
Group	Geographical
Importance	Medium

Notes	For a point-like place, this provides the longitude of the point. Note that it's the signed numeric representation, and not a string with *E* or *W* at the end. Note that the name is full, and not the abbreviation *lon* or *long*.
To publish	Publish this only in conjunction with a related `latitude` value.
To consume	N/A
See also	`latitude`

mediaType

Type	`Object` or `Link`
Value	An internet media type as defined in RFC 6838. Singular. Here are examples: • `text/html` • `image/png` • `application/postscript`
Group	Link
Importance	High
Notes	For a `Link`, this is the media type of the resource at the link's `href` URL. It's useful for differentiating `Link` objects by media type, so you can choose a PNG or JPEG version of the same image, for example. For an `Object`, this is the media type of the `content` property. This is by default `text/html` (HTML text), but could be other text formats, which is unusual, or even text encodings of binary data. However, as of this writing, very few (if any) ActivityPub applications support non-HTML content in the `content` property.
To publish	• Include a `mediaType` on `Link` objects when possible. • Avoid `mediaType` on `Object` types.
To consume	• In an array of `Link` objects, you can choose one based on the media types your application supports. • On `Object` objects, ignore or hide the `content` if the `mediaType` is not recognized.
See also	`source`

name

Type `Object` or `Link`

Value A string, singular, like the following:

- `Evan Prodromou`
- `ActivityPub: Programming for the Social Web`
- `Montreal`
- `ap`

This property can also be a language-tagged string, as follows:

```
"name": {
  "@value": "ActivityPub: Programming for the Social Web",
  "@language": "en"
}
```

Group Fundamental

Importance High

Notes This is a creator-provided name for the object. It's the primary text representation of the object.

To publish

- If possible, use the `nameMap` property. Using this property as a fallback can help for some consumers.
- Don't include HTML in this property.
- If the creator of the object has not provided a `name`, use an automatically created `summary` instead.
- Avoid the language-tagged string alternative, since it's not well supported by many ActivityPub consumers.

To consume

- If a `nameMap` is available, prefer it to this property.
- HTML-encode this property before including it in a web page. (Your web framework may do this for you.)
- Support the language-tagged string format. (This is hero-level ActivityPub.)

See also

- `summary`
- `nameMap`

nameMap

Type `Object` or `Link`

Value A JSON object, mapping language tags as defined in RFC 5646 to name strings, as in the following:

```
"nameMap": {
  "en": "England",
  "fr": "Angleterre"
}
```

The language tags can be fine-grained, including dialect differences between countries:

```
"nameMap": {
  "en-GB": "Cluedo",
  "en-US": "Clue"
}
```

Group Fundamental

Importance High

Notes This is the preferred way to specify language-specific names in ActivityPub. The `nameMap` property includes creator-provided names in as many languages as are needed, as described in Chapter 2.

To publish

- Use the `name` property as a fallback for consumers that don't understand the `nameMap` property.
- Don't include HTML in this property.
- If the creator of the object has not provided a name, use an automatically created `summaryMap` instead.

To consume HTML-encode the string values of this property before including them in a web page.

See also

- `summary`
- `name`

next

Type `Collection`

Value `id` of a `CollectionPage` or `OrderedCollectionPage`, singular

Group Collections

Importance High

Notes	This property is defined for all `Collection`types, but it makes sense only for `CollectionPage` or `OrderedCollectionPage`. It refers to the next collection page in the collection. One common way to loop through all the pages in a collection is to start with the collection's `first` property, and then follow each page's `next` property until there are no more `next` properties. In a reverse-chronologically sorted `OrderedCollection`, the `next` page will contain items added to the collection earlier. This can be really hard to keep track of!
To publish	• Include a `next` property in each collection page representation if you can. • The property can be in `id` representation.
To consume	Be prepared for `id` representation or object representation.
See also	• `prev` • `first`

object

Type	`Activity` or `Relationship`
Value	`id` of an `Object` or `Link`, usually singular but can be plural
Group	Activity
Importance	High
Notes	This represents the "direct object" of the activity—the object that was acted upon. Most activity types have an `object`, except those that derive from `IntransitiveActivity`. The indirect object is the `target` property. In the `Relationship` type, `object` is the object related to.
To publish	Avoid the plural form, since many consumers won't be expecting it.
To consume	• Be prepared for plural values. • Be prepared for `Link` values.
See also	• `actor` • `target`

oneOf

Type `Question`

Value `id`, usually an array of available responses, like the following:

```
[
   {
     "id": "https://social.example/note/3",
     "type": "Note",
     "name": "Gold"
   },
   {
     "id": "https://social.example/note/4",
     "type": "Note",
     "name": "Silver"
   },
   {
     "id": "https://social.example/note/4",
     "type": "Note",
     "name": "Lead"
   }
]
```

Group Question

Importance Medium

Notes This property allows exclusive-answer polls and questions. Responders can reply with only one of the defined answers.

To publish For maximum interoperability, use an array with `Note` objects, each of which has a unique `name` property.

To consume

- Watch for single values, which can happen if the poll is compacted.
- Watch for URL values.

See also `anyOf`

origin

Type `Activity`

Value `id` of an `Object` or `Link`, usually singular but can be plural

Group Context

Importance Medium

Notes	This property identifies an object from which the activity occurs. For example, in a `Move` activity, the `object` is moved from the `origin` to the `target`.
To publish	Avoid the plural form, since some consumers will not be able to handle it.
To consume	• Watch for plural values. • Watch for `Link` values.
See also	`target`

outbox

Type	`Object`
Value	`id` of an `OrderedCollection`, usually singular
Group	Actor
Importance	High
Notes	This is the collection of all activities performed by an actor, in reverse-chronological order. Reading from and posting to the outbox is described in Chapter 3.
To publish	• Some consumers are unable to use the object form of this property, so use a URL instead. • Some consumers use this property to duck-type an ActivityPub actor, so include it in actor objects, even if it can't be read or written to.
To consume	Be prepared for the object form.
See also	`inbox`

partOf

Type	`CollectionPage`
Value	`id` of a `Collection` or `OrderedCollection`, singular
Group	Collections
Importance	Medium
Notes	This identifies the collection that a collection page is part of.
To publish	The `id` representation, just a URL, should be fine for most applications.

To consume	Be prepared for the object form.
See also	`inbox`

preferredUsername

Type	`Object`
Value	A string
Group	Actor
Importance	High
Notes	This is a username for the user. The ActivityPub specification says that the name is not unique, but typically the name is unique relative to the domain of the actor `id`. This is used to construct the WebFinger identity for the actor, as discussed in Chapter 4. The property name is originally from the Portable Contacts standard, where it was used to indicate the username that someone would usually try to register when signing up for a new service. However, it was adapted to be the current username of the user; the "preferred" part is vestigial.
To publish	Use a plain string, not a language-tagged string.
To consume	N/A
See also	`name`

prev

Type	`Collection`
Value	`id` of a `CollectionPage` or `OrderedCollectionPage`
Group	Collection
Importance	Medium
Notes	This property is defined for all `Collection` types, but it makes sense only for `CollectionPage` or `OrderedCollectionPage`. It refers to the previous collection page in the collection. One way to loop through all the pages in a collection, which is somewhat unusual, is to start with the collection's `last` property and then follow each page's `prev` property until there are no more `prev` properties. In a reverse-chronologically sorted `OrderedCollection`, the `prev` page will contain items added to the collection later.

To publish	• Include a `prev` property in each collection page representation if you can. • The property can be in `id` representation.
To consume	Be prepared for an object representation or an `id` representation.
See also	`next`

preview

Type	`Object`
Value	`id` of an `Object` or `Link`, usually singular
Group	Image
Importance	Low
Notes	This property can be helpful for providing a thumbnail or preview of a binary object that is large and costly to download. For example, a `Video` object may have a `preview` that is a short trailer or excerpt of the video. For screenshots from a video, use `image` instead.
To publish	Include a `preview` property when possible for long-form video content.
To consume	• Be prepared for an object representation or an `id` representation. • Be prepared for a `Link` or an `Object`.
See also	• `icon` • `image`

published

Type	`Object`
Value	A datestamp, singular
Group	Fundamental
Importance	High
Notes	This is the creation timestamp for the object. It's useful for ordering objects chronologically.
To publish	Include a `published` property for all objects when available.
To consume	Use this property as a fallback for `updated` if it's not provided.
See also	`updated`

radius

Type	`Place`
Value	A nonnegative floating-point number, like `25.5`
Group	Geographical
Importance	Low
Notes	For a `Place`, this is the radius of a circle around the `latitude` and `longitude` value that defines the place. For example, a circle of about 30 kilometers around 45.54965, –73.87616 might define the Montreal urban region. The units are meters unless the `units` property of the place is also set.
To publish	Include a `radius` for places that are roughly circular and not pointlike.
To consume	Use this property as a guide for the rough dimensions of the place.
See also	• `units` • `altitude`

rel

Type	`Link`
Value	A link relation name from the Link Relation Registry. Here are examples: • `me` (another object that is the same as this object) • `author` (creator of this object) • `about` (the object that this object describes)
Group	Link
Importance	Medium
Notes	This is a code that defines the relationship with the resource at the `href` of the `Link`. `Link` relations are important in some circumstances, but often that information is conveyed by the context of the link. For example, if the `Link` is the `icon` property of an object, using `icon` as the `rel` property of the link is nice but not strictly necessary.
To publish	Include a `rel` property when the relationship to another object is well-defined and useful.

To consume	Treat the `rel` property as part of the uniqueness of a `Link`. Two `Link` objects are not the same if their `href` property is the same but their `rel` properties are different.
See also	`href`

relationship

Type	`Relationship`

Value	An `Object`, sometimes plural, usually as a URL. For example:
	`"relationship": "http://purl.org/vocab/relationship/siblingOf"`
	This property can also be plural:
	`"relationship": [`
	`  "http://purl.org/vocab/relationship/siblingOf",`
	`  "http://purl.org/vocab/relationship/colleagueOf"`
	`]`

Group	Relationship
Importance	Low
Notes	This is the type of relationship between the `subject` and the `object`. A taxonomy of relationships is not built into ActivityPub; the Relationship Vocabulary has a basic set for many relationships between individual humans. The Relationship group of types and properties, as of this writing, is not widely used on the social web.
To publish	• Include a `relationship` property with every `Relationship`. • Use the full URL of the relationship as the value. • Avoid the plural form; instead, use multiple `Relationship` objects, each with a different `relationship` property.
To consume	Be prepared for the plural form.
See also	• `subject` • `object`

replies

Type `Object`

Value A single `Collection` or `OrderedCollection`. For example:

```
"replies": {
  "id": "https://social.example/note/335/replies",
  "type": "OrderedCollection",
  "summary": "Replies to a note",
  "totalItems": 2,
  "orderedItems": [
    "https://social.example/note/483",
    "https://discussion.example/note/6944"
  ]
}
```

Group Reactions

Importance High

Notes This is the collection of replies to, or comments on, the object. See Chapter 3 for how this collection is created. Note that it includes the objects with `inReplyTo` equal to the containing object, not activities. If it is an `OrderedCollection`, it should be in reverse-chronological order.

To publish

- Use the object representation of the collection.
- Use an `OrderedCollection`.
- Include an `id`, `type`, and `summary`.
- Include `totalItems` so that consumers can quickly show the total number of replies.
- Filter the collection based on the consumer's authorization level. Don't include replies that the consumer is not authorized to read.
- If you can't determine their authorization level, include objects in the collection by URL, not in object representation.

To consume

- Be prepared for object or `id` representations.
- The collection will be filtered, so `totalItems` is likely to be a maximum value, not an accurate count of items you can read.
- As with most collections, the items may be in the collection itself or in collection pages.

See also

- `likes`
- `shares`
- `inReplyTo`

result

Type `Activity`

Value An `Object` or `Link`, possibly plural. For example, the result of a `Travel` activity may be a review of the destination:

```
"result": {
  "id": "https://social.example/travel/335/result",
  "type": "Note",
  "content": "I really enjoyed this destination."
}
```

Group Context

Importance Low

Notes This property defines direct effects or side effects of an activity. Many of the activity types that are widely used have the `object` as the most visible effect of the activity—for example, a `Create` activity results in an `object` being created.

`result` is not used for the core ActivityPub activity types, so it's mostly useful for extensions. I think it's an interesting part of the contextual framework of Activity Streams, but its full potential has not (yet?) been tapped.

To publish Use this property as an informative add-on.

To consume

- Be prepared for the `result` property to have plural or singular form.
- Be prepared for `Link` values.

See also

- `object`
- `target`

shares

Type `Object`

Value A single `Collection` or `OrderedCollection`. For example:

```
"shares": {
  "id": "https://social.example/note/335/shares",
  "type": "OrderedCollection",
  "summary": "Shares of a note",
  "totalItems": 3,
  "orderedItems": [
    "https://social.example/announce/221",
    "https://discussion.example/share/54",
    "https://forum.example/post/12567"
  ]
}
```

Group Reactions

Importance High

Notes This is the collection of `Announce` activities with the containing object as their `object` property. Note that it is a collection of activities, not actors.

Note that there is no `shared` collection in parallel with the `liked` collection for actors.

To publish

- Use the object representation of the collection.
- Use an `OrderedCollection`.
- Include an `id`, `type`, and `summary`.
- Include `totalItems` so that consumers can quickly show the total number of replies.
- Filter the collection based on the consumer's authorization level. Don't include activities that the consumer is not authorized to read.
- If you can't determine their authorization level, include objects in the collection by URL, not in object representation.

To consume

- Be prepared for object or `id` representations.
- The collection will be filtered, so `totalItems` is likely to be a maximum value, not an accurate count of items you can read.
- As with most collections, the items may be in the collection itself or in collection pages.

See also

- `likes`
- `replies`

source

Type `Object`

Value An `Object`, usually singular, typically with its own `content` and `mediaType`. For example:

```
"source": {
  "content": "@user@example.net https://evanp.me/ #cool",
  "mediaType": "text/plain"
}
```

Group Content

Importance Low

Notes This property provides the original source material used to generate the `content` of the containing object. It can be used by client software to make it easy to edit the object. The primary use case is for converting a text markup language, like Markdown, into HTML5.

One common markup language, the social microsyntax that includes links, @-mentions, and hashtags, is described in Chapter 3. It does not have an internet media type, but `text/plain` is a reasonable choice here.

This property does not have a multilanguage `*Map` version, so it's inappropriate for objects that use `contentMap` with multiple languages.

To publish

- Include this property primarily for the creator of an object.
- Include this property if the `content` property is used or if the `contentMap` property is used with a single language.

To consume

- Use this property for plain-text editing interfaces. Note that the client is still responsible for converting the `source` into the resulting format (usually HTML5) before submitting to the server.
- In the absence of this property, or if it uses an unfamiliar markup language, directly editing HTML5 is an acceptable fallback.

See also

- `content`
- `contentMap`

startIndex

Type	`OrderedCollectionPage`
Value	A nonnegative integer, singular: `"startIndex": 100`
Group	Collections
Importance	Low
Notes	This provides a hint of the page's location in the order of pages for the `OrderedCollection`. Note that the index is for the first item in the `orderedItems` or `items` array, not the page number. This property can be helpful for navigating any `OrderedCollection` where new elements are generally inserted at the end. However, in ActivityPub, collections are mostly ordered reverse chronologically, so that new items are inserted at the beginning. That makes these index numbers unstable and less useful for navigating a collection.
To publish	You can safely leave this property out of ActivityPub collections.
To consume	• Use this property only as a hint of the page's location in the collection. • Use `next` or `prev` to navigate through a series of pages.
See also	• `object` • `target`

startTime

Type	`Object`
Value	A datestamp, singular: `"startTime": "2024-09-01T00:00:00Z"`
Group	Time
Importance	Medium
Notes	For objects with extent in time, this determines the beginning time of the object. An `Event`, for example, will usually have a `startTime` and a `duration` or an `endTime`. Some activity types, like `Travel`, `View`, or `Read`, have an extent in time and would benefit from a `startTime` property.

To publish	• Use UTC timestamps if possible. • Don't include fractions of a second, if possible.
To consume	• Many libraries exist for parsing this timestamp format, ISO 8601, into your programming language's internal representation. • The object's publication schedule may be unrelated to the start and end time of the event described. Don't try to compare or harmonize `published` and `updated` with `startTime`.
See also	• `endTime` • `duration`

streams

Type	`Object`
Value	An array of `Collection` or `OrderedCollection` objects, in object representation or `id` representation:

```
"streams": [
  {
     "id": "https://social.example/user/8/priority",
     "type": "OrderedCollection",
     "summary": "Priority inbox for Evan"
  },
  {
     "id": "https://social.example/user/8/spam",
     "type": "OrderedCollection",
     "summary": "Spam folder for Evan"
  },
  {
     "id": "https://social.example/user/8/stories",
     "type": "OrderedCollection",
     "summary": "Stories from Evan's friends"
  }
]
```

Group	Actor
Importance	Low

Notes

I want to like the `streams` property. It provides some extensibility so that the actor or others can view related streams beyond the main `inbox`, `outbox`, `followers`, `following`, and `liked`. Its description in the ActivityPub specification is "A list of supplementary Collections which may be of interest," which sounds cool and mysterious, like a folder that James Bond would receive from M.

But it's not clear how to programmatically find collections that may be useful. If my ActivityPub client supports making contact lists or watching stories, there doesn't seem to be a way to find that kind of collection in the `streams` property. It's also not possible to determine the type of object in each stream.

The best solution I can think of is using an extended type (`Contact List`) or including an extension flag in the collection listing (`"isContactList": true`).

Adding more-specific streams to the ActivityPub client experience is important, but I think more useful when implemented as top-level extension properties for the actor. However, it may be useful to also include those collections in `streams`, in case a client lets the user manually navigate them.

To publish

- Prefer extension properties of an actor to this collection.
- Include extension collections in this array, in hopes that clients will let the user review them manually.

To consume

Provide a way to navigate these streams manually. This may require a general-purpose stream interface, since there is no indicator of the type of object in each stream.

See also

- `inbox`
- `outbox`
- `endpoints`

subject

Type

`Relationship`

Value

A single `Object` or `Link`, as in the following:

```
"subject": {
  "id": "https://social.example/users/45127",
  "name": "Phyllis D.",
  "type": "Person"
}
```

Group

Relationship

Importance Low

Notes In a `Relationship`, this is the object that is the origin of the relationship. It is not necessarily the creator of the relationship.

Note that this property is not used in activities; use `actor` for the "subject" of the sentence there.

To publish Use a functional representation of the subject.

To consume
- Watch for `id` representations.
- Watch for object representation, and reuse data if possible.
- Watch for `Link` objects.

See also
- `object`
- `relationship`
- `actor`

summary

Type `Object`

Value A string, summarizing an object:

```
"summary": "A note by Evan"
```

The value can also be a language-tagged string:

```
"summary": {
   "@value": "A note by Evan",
   "@language": "en"
}
```

Group Fundamental

Importance High

Notes This is a brief description of the object—for example, an abstract of an `Article` or the text description of an `Image`. It can be provided by the creator or can be automatically generated by the ActivityPub client or server. It should be about a paragraph or less.

The summary can include HTML5 markup.

When a `name` or `nameMap` is not available, the `summary` is used as the primary textual representation of the object. If this is the case, the summary should be short, at most a sentence, and not include HTML5 markup.

For Mastodon and some other ActivityPub software, this property is used as a content warning for potentially sensitive text, images, or video.

To publish Use `summaryMap` instead of the language-tagged string format.

To consume
- Be prepared for the language-tagged string format.
- Sanitize the HTML in this property before including it in a web page or other HTML processor.

See also
- `summaryMap`
- `name`
- `content`

summaryMap

Type `Object`

Value A JSON object mapping language tags as defined in RFC 5646 to the content of the object, such as the following:

```
"summaryMap": {
  "en": "A note by Evan",
  "fr": "Une note d'Evan"
}
```

Group Fundamental

Importance High

Notes See the description of `summary` for how this property is used.

This alternative allows multiple summaries in different languages.

To Publish
- If the creator provides multiple language versions of the summary, include them in the `summaryMap`.
- If the summary is automatically created by client or server software, provide it in one language with an appropriate language tag. Automated translation is best left up to the consumer.

To consume
- Choose the right language tag based on the user's language preferences. RFC 4647 outlines some tools and techniques for matching languages.
- Sanitize the HTML5 in the summary value before displaying in a web page.

See also
- `summary`
- `name`
- `content`

tag

Type `Object`

Value An array of `Object` or `Link` values, such as the following:

```
"tag": [
  {
     "type": "Mention",
     "href": "https://social.example/user/175",
     "name": "@jeff@social.example"
  },
  {
     "type": "Hashtag",
     "href": "https://tags.example/poutine",
     "name": "#poutine"
  }
]
```

Group Content

Importance High

Notes This property is for objects related through tagging, both as a classification technique and as a way to note someone's or something's relationship to the object.

To publish

- Use the functional representation of all the tagged objects.
- Use a `Mention` to tag an actor.
- Use a `Hashtag` for inline or out-of-band categorization.
- Include a `Place` if it is the topic discussed in the object. Otherwise, use `location`.

To consume

- Be prepared for a single value, because of JSON-LD compaction.
- Be prepared for both `Object` and `Link` types, including `Mention` and `Hashtag`.

See also `attachment`

target

Type	`Activity`

Value	`Object` or `Link` value, sometimes plural:
	`"target": {`
	`  "type": "Collection",`
	`  "id": "https://social.example/note/13/replies",`
	`  "name": "Replies to note 13"`
	`}`

Group	Activity
Importance	Medium
Notes	This is usually for the "indirect object" of an activity—the direction of motion in the activity. For the `Add` type, for example, this property identifies the `Collection` the `object` was added to.
To publish	• Use a brief representation or more. • Avoid using plural values.
To consume	• Can have a plural value • Can be a `Link` object
See also	`origin`

to

Type	`Object`

Value	`Object` or `Link` value, often plural:
	`"to": [`
	`  "https://social.example/users/17",`
	`  "https://photos.example/person/97",`
	`  "https://blog.example/user/25/followers"`
	`]`

Group	Addressing
Importance	High
Notes	This is the primary, public addressing property. It's used for authorization in reading and reacting to objects and activities, as well as for delivery of activities locally and remotely.
To publish	Use `id` representation (just a URL) to maximize portability. Not all consumers can handle an object representation here.

To consume	• Can have a plural or single value • Can include object representation or `id` representation • Can include `Link` objects
See also	• `bto` • `cc`

totalItems

Type	`Collection`
Value	A nonnegative integer, singular, such as `35`
Group	Collections
Importance	Medium
Notes	This is a hint at the total number of items in the collection. Because of filtering, or for other reasons, not all those items may be readable.
To publish	Don't include this property on `CollectionPage` or `OrderedCollectionPage` objects.
To consume	Represents a maximum number of visible items.
See also	• `items` • `orderedItems`

type

Type	`Object` or `Link`

Value	A name from the AS2 vocabulary, or a prefixed name from an extension vocabulary, or a URL. Possibly plural. Here's an example:
	`"type": "Person"`
	`"type": [ "Person", "vcard:Person", "https://schema.org/Person" ]`

Group	Fundamental
Importance	High

Notes This property indicates the type, possibly plural types, of the object. It is fundamental to the way AS2 and ActivityPub work. With rare exception (the `endpoints` property, for example), every JSON object in an AS2 document should have a `type` property.

To publish Use at least one type from the AS2 vocabulary for each object, even if it's only `Object`.

To consume
- This property can have plural values.
- If this property is not provided, use `Object` as the default.
- Be careful with comparisons! The bare name (`Person`), prefixed name (`as:Person`), and full URL (`https://www.w3.org/ns/activitystreams#Person`) are equivalent.

See also `id`

units

Type `Place`

Value One of the following units of distance:

- `cm` (centimeters)
- `feet`
- `inches`
- `km` (kilometers)
- `m` (meters)
- `miles`

Or, alternately, a URI identifying an extension unit type

Group Geographical

Importance Low

Notes These are the units used for both `altitude` and `radius`.

To publish
- Include this property if your `Place` has an `altitude` or `radius`.
- Just use the default value, `m` (meters). C'mon.

To consume
- If a publisher uses a unit besides meters, just grit your teeth and convert it into meters.
- If a publisher insists on making your life difficult by using an extension unit type like parsecs or furlongs or angstroms, just close your laptop and throw it into the ocean. There are better ways to live.

See also

- `altitude`
- `radius`

updated

Type `Object`

Value Timestamp, singular, like `2024-05-27T00:00:00Z`

Group Fundamental

Importance High

Notes This is the date at which the object itself was last modified.

Note that this is unrelated to changes in objects referenced by the properties of the object. For example, this object's `updated` property is different from the `updated` property of the `attributedTo` actor:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "id": "https://social.example/user/38/note/18",
  "type": "Note",
  "to": "as:Public",
  "content": "Hello, World!",
  "updated": "2024-05-01T00:00:00Z",
  "attributedTo": {
    "id": "https://social.example/user/38",
    "name": "Nick M.",
    "updated": "2024-05-14T00:00:00Z"
  }
}
```

To publish Include this property for the link representation or full representation of an object.

To consume If `updated` is not available, assume `published` is the latest change timestamp for the object.

See also `published`

url

Type `Object`

Value One or more URLs or `Link` objects. For example:

```
"url": "https://social.example/uploads/file.png"
```

Here's the `Link` option:

```
"url": [
  {
    "type": "Link",
    "mediaType": "image/jpeg",
    "href": "https://social.example/uploads/file.jpg"
  },
  {
    "type": "Link",
    "mediaType": "image/webp",
    "href": "https://social.example/uploads/file.webp"
  }
]
```

Group Fundamental

Importance High

Notes For binary file types, like `Image`, `Video`, or `Audio`, this property represents one or more locations for the binary data for the object—often differentiated by format or size.

For ActivityPub actors, this is used for the profile page of the actor, usually in HTML format.

For other types, this is a link to a representation of the object.

It's possible, but risky, to include other URL protocols besides HTTPS. For example, for small binary files that you want to transfer in full to save on HTTP traffic, you could try including a `data:` URL, defined in RFC 2397, which encodes the file in the URL string itself. Other options might include using URLs for peer-to-peer networks with content addressing, like IPFS.

If you include other protocols, do it carefully. Include an HTTPS URL as a fallback, preferably early in the array of alternative `Link` objects. Most ActivityPub consumers won't recognize the format and may just throw an error if they encounter a URL type they don't recognize.

To publish	• Use the `Link` format where possible. • Provide the `mediaType` for most representations; use `text/html` for a web page. • Provide `height` and `width` for images and video. • Don't use a `rel` value unless it's `alternate` (an alternate representation of the current object). • If you include links with different protocols, include at least one link with the `https:` protocol as a fallback.
To consume	• Choose among the options by using the metadata in the `Link` objects. • In the absence of `Link` objects, you may be able to retrieve additional metadata by using HTTP HEAD requests to the URLs provided.
See also	• `alsoKnownAs` • `href`

width

Type	`Link` (not `Object`!)

Value	Nonnegative integer, as in the following:
	`"url": {`
	`  "type": "Link",`
	`  "width": 100,`
	`  "height": 100,`
	`  "mediaType": "image/png",`
	`  "href": "https://domain.example/image.png"`
	`}`

Group	Link
Importance	Medium
Notes	Frustratingly, this property is only for `Link` objects. The best way to use it for an `Image` or `Video` is to have a `Link` object for the `url` property, and include the height (and width) there.

To publish
- Only for `Link` objects.
- Including multiple `url` properties with different heights and widths can be helpful for clients picking an appropriately sized image.

To consume If a `Link` is in an array, you can sometimes pick the best size image or video by comparing `height` and `width`.

See also `height`

About the Author

Evan Prodromou is the Research Director of the Social Web Foundation. He is the coauthor of the ActivityPub protocol and the Activity Streams 2.0 data format. Sometimes called "The Father of the Fediverse," Evan made the first-ever post on the social web in May 2008. He founded the identi.ca website and StatusNet software (now GNU Social), as well as coauthoring the OStatus social-network specification. As chair of the W3C's Social Web Working Group, he coordinated the development of ActivityPub and AS2 into official standards. He won the O'Reilly Open Source Award in 2009 as "Best Social Networking Hacker." Evan lives in Montreal, Canada, with his wife and two children, where he works on open source software to fight climate change. He loves hacking the ActivityPub API and protocol, cooking Greek and Palestinian food, and gardening with North American native plants.

Colophon

The animal on the cover of *ActivityPub* is a nanday conure (*Aratinga nenday*), also known as a nanday parakeet or black-hooded parakeet.

Nanday conures have a striking appearance. They are predominantly green in color with a gray-blue chest, blue tail feathers, and splash of red on their legs. Their black faces and beaks are their most distinguishing feature. Adults are 11–12 inches long and weigh approximately five ounces.

Nanday conures are native to South America and can be found in Argentina, Bolivia, Brazil, and Paraguay. They usually make their nests in holes in trees. Their diet consists of berries, fruit, seeds, and flowers.

Although energetic and loud, these birds are popular pets. Affectionate toward their owners and highly intelligent, they are capable of learning several words and may learn to talk as well as perform tricks.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from a loose plate, source unknown. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.