Beyond Scripting

Building Swift Apps for Apple Administrators

Charles Edge
Joel Rennich
Jeremy Bannister

Apress®

Beyond Scripting

Building Swift Apps for Apple Administrators

Charles Edge
Joel Rennich
Jeremy Bannister

Apress®

Beyond Scripting: Building Swift Apps for Apple Administrators

Charles Edge
Minneapolis, MN, USA

Joel Rennich
Stillwater, MN, USA

Jeremy Bannister
New York, NY, USA

ISBN-13 (pbk): 979-8-8688-3034-1
ISBN-13 (electronic): 979-8-8688-3035-8
https://doi.org/10.1007/979-8-8688-3035-8

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Miriam Haidara
Editorial Project Manager: Marina Engler

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a Delaware LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

To Charles, thank you for bringing us into
the crazy mess that is writing a book.
Jeremy and Joel

Table of Contents

About the Authors

Charles Edge was the CTO of bootstrappers.mn and the CTO/COO of secretchest.io and a former director at Jamf, where he led the development of Jamf Now. He held 35 years of experience as a developer, administrator, architect, product manager, entrepreneur, and CTO. He authored 25 books and more than 6,000 blog posts on technology and served as an editor and author for a number of publications. Charles served on the board of directors for a number of companies and nonprofits and frequently spoke at conferences including DefCon, BlackHat, LinuxWorld, the Apple Worldwide Developers Conference, and a number of Apple-focused conferences. Charles also authored krypted.com and was a cofounder/host of the MacAdmins Podcast and The History of Computing podcast. He passed away in April of 2024, and his extensive network of people felt the loss very deeply.

Joel Rennich is the Senior Vice President of Product at JumpCloud and resides in the greater Minneapolis, MN, area. While Joel has spent most of his professional career focused on Apple products, at JumpCloud, he works with a team focused on boldly going where no devs have gone before across all platforms. In 2018, Jamf acquired Orchard & Grove, a start-up he co-founded, which is where Joel developed the widely used open source software NoMAD. Over the years, Joel has been a frequent speaker at a number of conferences including WWDC, MacSysAdmin, MacADUK, Penn State MacAdmins Conference, Objective by the Sea, FIDO Authenticate, and others in addition to user groups everywhere. Joel spent over a decade working at Apple in Enterprise Sales and started the website afp548.com, which was the mainstay of Apple system administrator education during the early years of macOS X.

Jeremy Bannister is a Swift developer from New York who now lives in Spain. He connected with Charles Edge in December of 2021 and began working for Charles to prototype a password management tool with a focus on resisting quantum computing decryption attacks. They worked together until Charles' unexpected passing in April 2024. Jeremy is happy to have helped proofread Charles' manuscript and helped to allow his final book to be published.

Acknowledgments

The authors would like to recognize Apple for all of its work making Swift an approachable coding language that allows non-developers like us to bring our ideas out into the light. It's been amazing to see Swift get more robust and powerful as the years have gone by.

We appreciate the Mac Admin Community, which, as always, is an amazing place to see an idea come to fruition. A number of members in the community have helped nudge this book along over the too lengthy chunk of a time that it took us to finish. From heated conversations over lunch in the Chalmer's cafeteria during MacSysAdmin to too-numerous-to-count conversations on the MacAdmins Slack, we appreciate all you have done to help us think through this.

And we send thanks to our readers. We look forward to seeing what you bring into this world as a result of using this text. For us, bringing an application into existence is a delight, and we hope this work helps you experience that joy.

Introduction

System administrators occupy a unique role in the world of technology. We are intimately familiar with the systems we manage, not just how to configure them but how they feel when something goes wrong at 4 a.m., the dread of a failing script in production, the satisfaction of a Friday without any tickets because a deployment worked. For most of our careers, we've been the ones who keep things running so the developers can keep building.

This book is an invitation to cross that line.

Swift was designed by Apple to be approachable – a language for everyone who wants to build for the Apple ecosystem, not just people with computer science degrees. That premise aligns almost perfectly with the MacAdmins community, where many of the best minds in the business are self-taught, lateral thinkers who learned by doing and by sharing. The MacAdmins world has always run on open source, on blog posts, on "here's how I figured it out" – and that spirit is very much alive in these pages.

This book will not make you a full-time software developer. If that's what you're after, there are plenty of resources that go much deeper into theory and architecture. What this book will do is get you dangerous – dangerous in the best possible sense. It will give you the tools to build the things you've been imagining: a small utility to tell users what's happening during a management workflow, an authentication helper that fits your environment exactly, a tiny automation that lets you enjoy your morning coffee a bit more. The best way to use this book is with a project in mind. It doesn't need to be ambitious – in fact, the less ambitious the better to start. Each chapter adds a piece, and by the end, the pieces fit together.

A Note on Xcode

Most of this book assumes you are working in Xcode, Apple's development environment. If you don't have it installed, now is a good time. It's free from the Mac App Store and brings everything you need to follow along with the examples in these chapters.

A Note on What This Book Isn't

This book doesn't try to be comprehensive. Swift is a deep language with a large ecosystem, and a thorough treatment of everything it offers would run to several volumes. What these pages try to do is give you a foundation that's complete enough to be useful, and honest about where the edges are.

You will hit those edges. You will encounter error messages that aren't covered here. You will find yourself Googling. That's not a failure – that's programming. The goal is that when you arrive at those moments, you'll have enough context to understand what you're reading and enough foundation to find your way through.

The community that formed around MacAdmins has always been a welcoming one, full of people who genuinely want to help the people who are learning. Don't be afraid to ask questions. That openness, that willingness to share what you know and admit what you don't, is one of the things that makes this community worth being part of.

Let's get started.

CHAPTER 1

Introduction to Swift App Deployment

This book has covered how to write code so far – much of which can work in the REPL or basic scripts. However, many admins will need to surface information to the screen, not just run a simple script. As these projects grow, it's important to be able to use Xcode to publish apps. The simplest way to think of publishing an app is to build it to run on the App Store. Or at least that's the simplest way for a user to access the app. But there are lots of options, with each moving the friction of getting an app to run more on the side of the person who's getting it and less on the developer's side. In this context, friction means the amount of effort required by the end user to install and run software, such as trusting certificates, installing profiles, or manually launching binaries.

The most difficult way to publish an app would probably be to post source code for an app with multiple entitlements to GitHub – without a compiled version. Somewhere in the middle, there is also a B2B store, TestFlight for members of an organization, TestFlight for people not in the organization, and distributing a compiled binary outside of the App Store, each with a little more friction to get installed than the last. With these distribution trade-offs in mind, let's look at how Xcode supports building and preparing apps for these different delivery methods.

Ways to Publish Apps

The options to publish an app can be seen in the Product menu within Xcode. One of the most common to use is the option to build and run. This is the most common and straightforward way to compile an app. Clicking the "Run" button in Xcode triggers a build process that includes

C. Edge et al., *Beyond Scripting*, https://doi.org/10.1007/979-8-8688-3035-8_1

- *Swift compilation setup*: Source files and conditional compilation blocks are resolved for the active build configuration.
- *Compiling*: Swift source files are compiled into object files.
- *Linking*: Object files are linked together to create a single executable file.
- *Running*: The executable file is launched on the simulator or connected device.

This method is convenient for quick testing and development iterations. It can be time-consuming for larger projects but produces a number of products (thus the name of the menu). In the Product menu, there's also an option for "Show Build Folder in Finder" and "Copy Build Folder Path". These can be used to locate the products that were created and potentially run them. Keep in mind that development builds usually aren't meant for broad distribution without proper distribution signing and provisioning.

The other options commonly used include Test, Profile, Analyze, and Archive:

- *Test*: Builds the target (the app bundle) for the purposes of running unit tests. For more on unit tests, see Chapter 7.
- *Profile*: Builds and runs the target for use in instrumentation. This can be used to isolate memory leaks, problems with allocations, or issues with compiler options.
- *Analyze*: Finds specific types of bugs that might block an app from being accepted for different forms of distribution.
- *Archive*: Builds the app for specific channels of distribution, which include the following:
  - **TestFlight** is for distribution through Apple's TestFlight program. This requires the setup for App Store Connect to be complete and is meant for releasing software for beta testing.
  - **Ad Hoc** is used to distribute an app to testers through a channel other than TestFlight. This might be via a .ipa file to devices registered with a provisioning profile.
  - **Enterprise distribution** is similar to Ad Hoc distribution but uses an Enterprise distribution certificate.

- **Development distribution** signs a build with development credentials for development/testing workflows (e.g., on registered test devices).

Before going further, it's common to try a few of these options over the course of development. This leads to cruft in the build folder (or old, out of date, icky junk). There's a button to "Clean Build Folder…" in the Product menu. Use that when there are weird errors or, as a general rule of thumb, when moving to the next phase of releasing software. There are options for schemes, scripts, localizations, CI/CD, and other aspects of software development, but we'll get to those later. Next, let's cover what goes into those target folders.

Targets

In Xcode, a target is a specific configuration or build setting that defines how a project or app is compiled, built, and packaged. It represents a distinct product or component within a project and provides a way to organize and manage different versions or variations of an app.

Xcode can create multiple targets within a project. Each target can represent a different version, platform, or configuration of an app – and might also include app extensions. Each of these targets can have their own set of build settings and configurations that determine how the code is compiled, linked, and bundled. Build settings include options like compiler flags, linker settings, deployment targets, code signing, and more.

Targets allow for customizing various aspects of an app, such as different app icons, display names, bundle identifiers, resource files, and build settings. This flexibility can be used to create different versions of the app, such as a free version and a paid version with additional features. Each target can have its own dependencies and frameworks. Each can include or exclude specific libraries, frameworks, or modules for different targets, based on their requirements.

As described earlier, Xcode also provides the ability to create test targets, which are dedicated targets for unit tests, UI tests, or other types of testing. Test targets have their own set of source files and configurations specifically for testing purposes.

Targets can have different deployment targets, indicating the minimum version of the operating system or platform on which the app can run. This means it can be used

to support different iOS versions or macOS versions for different targets. Developers can build and run specific targets individually within Xcode. This allows developers to focus on specific parts of a project, test different configurations, or create separate builds for different targets. Now, let's look at one of the easiest ways to get apps in the hands of users: TestFlight.

Running and Testing the App in the Simulator

Click the play button in Xcode (or if hovered over, the "Start the active scheme" button) and an app builds and runs. There are other options in the Product menu covered in this book, but when building for Mac, the app just opens. When building for iPhone, iPad, Apple Watch, Apple TV, or Apple Vision Pro, Apple includes a tool called the simulator to test apps with. The easiest way to work with the simulator is through Xcode. Click that same button for an iPhone app and the simulator will open and the app gets deployed. Simply quit the app to free up that memory and close the instance of the simulator.

It's also possible to use the command line tool simctl for interacting with the simulator. This is helpful in automating QA operations when possible. The simctl binary is located at /Applications/Xcode.app/Contents/Developer/usr/bin/simctl and typically accessed as a verb from the /usr/bin/xcrun command.

First, let's list all simulators using the `list` command by running `xcrun simctl list`:

```
/usr/bin/xcrun simctl list
```

The output shows device types, runtimes, and devices, most of which should show as unavailable or shutdown (the following list is truncated):

```
== Device Types ==
iPhone <Model> (com.apple.CoreSimulator.SimDeviceType.<DeviceTypeID>)
Apple Watch <Model> (com.apple.CoreSimulator.SimDeviceType.<WatchTypeID>)
== Runtimes ==
iOS <Version> (com.apple.CoreSimulator.SimRuntime.iOS-<Version>)
== Devices ==
-- iOS <Version> --
iPhone <Model> (<UUID>) (Shutdown)
```

Next, let's create a fresh new simulator called `testing_iPhone`. First, find an available device type and runtime:

```
/usr/bin/xcrun simctl list devicetypes
/usr/bin/xcrun simctl list runtimes
```

Then create the simulator with IDs that exist on your machine:

```
/usr/bin/xcrun simctl create testing_iPhone <DeviceTypeID> <RuntimeID>
```

The output includes a UUID such as the following. That can then be used to track further interactions with the simulation:

```
E6D4C2B8-0601-4557-99DA-B6B8251D534D
```

The most common tasks would be booting, shutting down, erasing, and opening simulators. First, let's boot it up:

```
/usr/bin/xcrun simctl boot E6D4C2B8-0601-4557-99DA-B6B8251D534D
```

To shut that same simulator down, use the shutdown verb:

```
/usr/bin/xcrun simctl shutdown E6D4C2B8-0601-4557-99DA-B6B8251D534D
```

Neither of these commands provides any output on success, but both return errors on failure. Once you've run tests, I like to erase my simulator and start fresh. To do so, simply use the erase command:

```
/usr/bin/xcrun simctl erase E6D4C2B8-0601-4557-99DA-B6B8251D534D
```

To open the Simulator app, you can use

```
open /Applications/Xcode.app/Contents/Developer/Applications/Simulator.app/
```

For more information on using simctl subcommands (or to see how they might have changed over time in case any of these don't work), use the help subcommand:

```
/usr/bin/xcrun simctl help
```

Notice there are a lot of verbs:
create: Create a new device.
clone: Clone an existing device.
upgrade: Upgrade a device to a newer runtime.
delete: Delete a device or all unavailable devices.
pair: Create a new watch and phone pair.
unpair: Unpair a watch and phone pair.

pair_activate: Set a given pair as active.
erase: Erase a device's contents and settings.
boot: Boot a device.
shutdown: Shutdown a device.
rename: Rename a device.
getenv: Print an environment variable from a running device.
openurl: Open a URL in a device.
addmedia: Add photos, live photos, videos, or contacts to the library of a device.
install: Install an app on a device.
uninstall: Uninstall an app from a device.
get_app_container: Print the path of the installed app's container.
launch: Launch an application by identifier on a device.
terminate: Terminate an application by identifier on a device.
spawn: Spawn a process by executing a given executable on a device.
list: List available devices, device types, runtimes, or device pairs.
icloud_sync: Trigger iCloud sync on a device.
pbsync: Sync the pasteboard content from one pasteboard to another.
pbcopy: Copy standard input onto the device pasteboard.
pbpaste: Print the contents of the device's pasteboard to standard output.
help: Prints the usage for a given subcommand.
io: Set up a device IO operation.
diagnose: Collect diagnostic information and logs.
logverbose: Enable or disable verbose logging for a device.

Let's look at the pasteboard commands as an example for automating more options. macOS has `pbcopy` and `pbpaste` for the Mac clipboard. `simctl` provides similar commands for a simulator device.

For example, the following copies text from a file into the *booted simulator* pasteboard:

```
xcrun simctl pbcopy booted < ~/Desktop/transfer.txt
```

Then read the simulator pasteboard back to standard output:

```
xcrun simctl pbpaste booted
```

Once the data is copied, the transfer file can then be removed:

```
rm ~/Desktop/transfer.txt
```

The `xcrun simctl` tool also comes with a number of other useful automation workflows.

Running an app on a Mac or in the simulator is great, but development runs are generally local testing workflows. Most will want their creations to run on more than just one device.

The Invisible Hand of Code Signing

As mentioned, development builds are controlled by code signing. Xcode uses certificates, entitlements, and provisioning to sign apps, verify origin, and enforce where a build can run. The private key stays on the signing machine and should never be shared.

Think of it like a handshake between an app and the operating system. If the signature, provisioning, or entitlements aren't valid for that platform and device, the app won't run. The easiest answer for broad public distribution is the App Store (or TestFlight for beta): Apple handles distribution signing and trust for end users. However, not all apps are meant for mass distribution.

Another option for iOS is to export a `.ipa` file (or a `.app` bundle for macOS) from Xcode. Installation is usually handled through Apple-managed channels such as TestFlight, MDM, Apple Configurator, or signed/notarized macOS distribution workflows.

Apps are compiled to ~/Library/Developer/Xcode/DerivedData/-/Build/Products/Debug-iphonesimulator for simulator builds by default in Xcode or ~/Library/Developer/Xcode/Archives for release builds. It's also possible to set a custom location for builds. To do so, click the File menu and select Project Settings and then Advanced. Modify the "Build Location" setting for "Debug" and/or "Release" builds to change where they're saved (Figure 1-1).

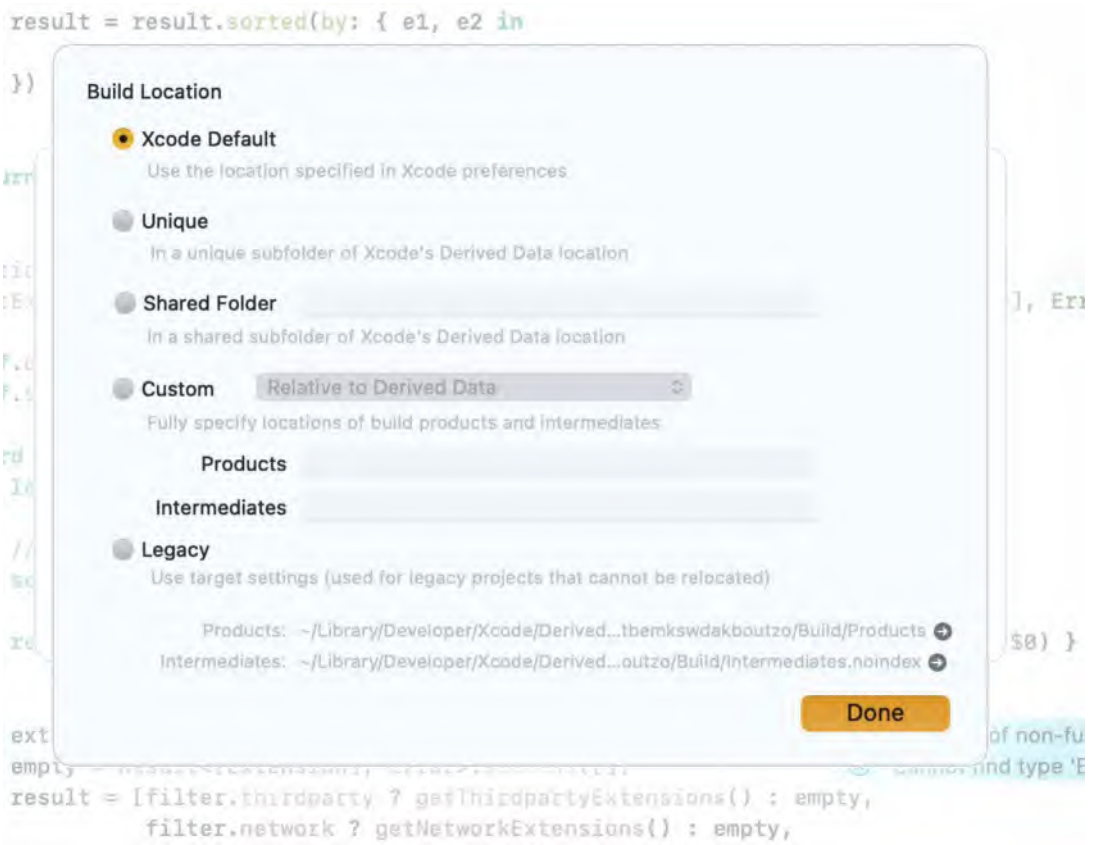

Figure 1-1. *Setting a build location in Xcode*

Keep in mind that simulator builds include a `.app` file and usually a `.dSYM` file that contains debugging information. For physical devices, Xcode installs an app bundle directly during development runs; a `.ipa` is typically produced when exporting an archive for distribution. The Derived Data folder stores temporary build files that can be cleared to free up space (`~/Library/Developer/Xcode/DerivedData`). This is also where a Mac app bundle will be found. It's also always possible to use the Project Navigator to access the "Products" folder for built products for quick access. Again, deployment to devices depends on the chosen signing/distribution method. Therefore, TestFlight is a great option to get apps to devices in a way with less friction (although there's still plenty of that).

Notarization

Notarization is the process where developers submit their macOS app to Apple, which then scans the binary for any malicious code or code-signing issue in an automated fashion. If the binary is found to be acceptable, Apple generates a ticket that can be "stapled" to the application and stored online by Apple. This ticket is used by Gatekeeper and the macOS kernel to validate the binary before launching it and ensure only proper software can be run.

By using Xcode to build your software, most of the notarization process will be taken care of for you. Note that notarization can take some time, although rarely more than few minutes, for Apple's servers to process your submitted binary.

If compiling outside of Xcode, you can use the `xcrun notarytool` command to help you manage the notarization process.

TestFlight

Simply put, TestFlight is an app on the App Store that can be used to install beta versions of software safely. There are two audiences for TestFlight: internal testers (App Store Connect users on your team) and external testers. Internal distribution does not require Beta App Review; external distribution does.

Choose Who Gets What

The distribution groups can be seen in the TestFlight tab for each app. Here, each version is listed along with the groups that the version has been distributed to. Figure 1-2 shows how versions that weren't distributed to an external group show as "Ready to Submit" and the first to go to an external group was Rejected (in this case because the export controls options hadn't been set) and the next version was accepted, so has a green checkmark.

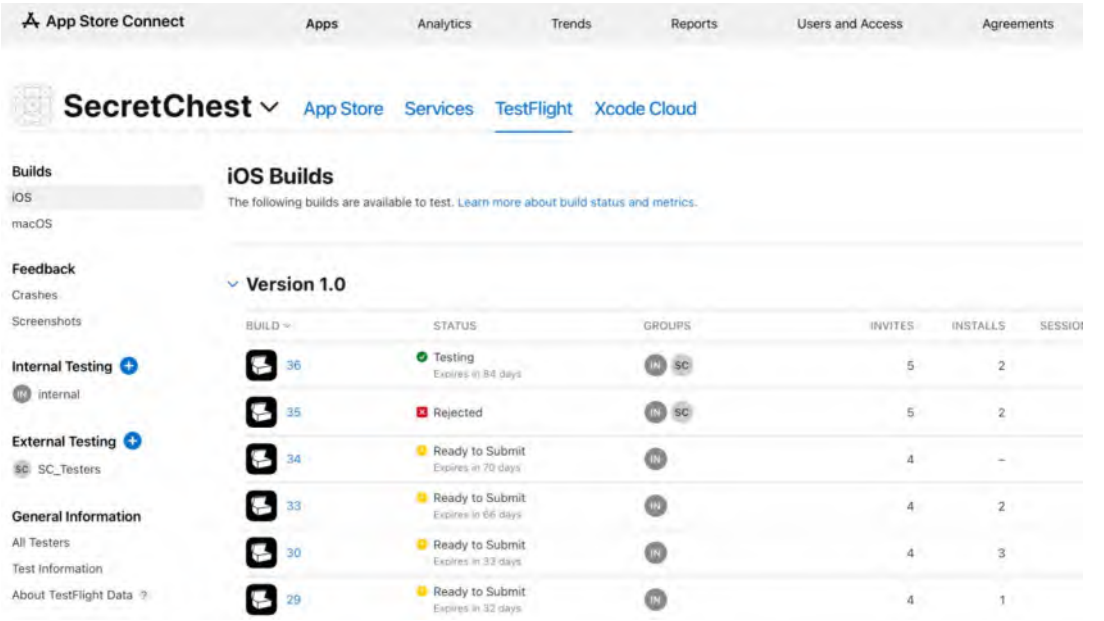

Figure 1-2. *Internal vs. external distribution*

Creating groups is done with the blue plus sign next to each type of group to be distributed to. Once a group is created, click on it in the sidebar and then scroll down to the bottom of the screen to add members. Click the plus sign in the last section to add a member (Figure 1-3).

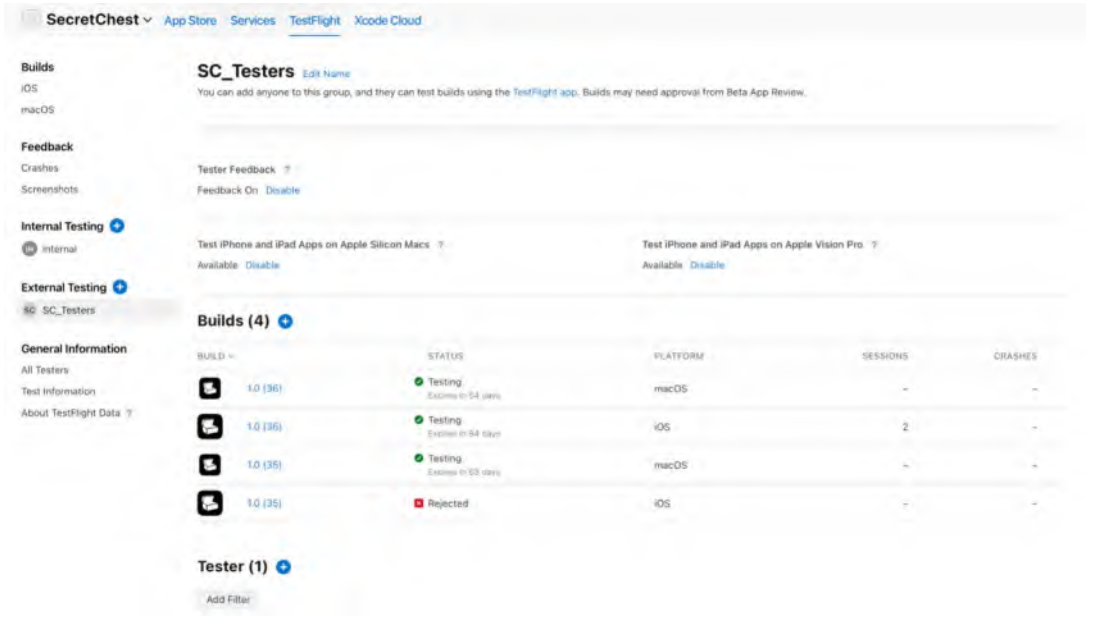

Figure 1-3. *Add users to distribution groups*

There are three options to add users to groups. Add Existing Testers allows a developer to add testers that have already agreed to work with the organization. Add New Testers allows for entering the email of a new user, who will get an email from Apple to join the group. Import a CSV is used for bulk adding (up to 100 internal testers and up to 10,000 external testers per app). Different groups can then be used for different versions that are at different levels of maturity as release cadences become more dogmatic.

Post an App to TestFlight

To post an app to TestFlight, first prepare the app, which is to say to make sure it runs in the simulator, compiles, is signed, and is ready for distribution. Test apps thoroughly to identify and fix any bugs or issues. Also, consider creating a release build specifically for TestFlight distribution.

Once the app is ready, it's time to log into App Store Connect at `https://appstoreconnect.apple.com/login`. To do so, developers need an Apple Developer account. Once logged in, select the "My Apps" section and choose the app to post to TestFlight from the list. Then click the "TestFlight" tab in the app's dashboard. Upload a build from Xcode Organizer (or Transporter), wait for processing to complete, and then add that processed build to tester groups.

Each build can also have different metadata that informs users about the build and helps developers triangulate build-centric issues that come up. Fill in the necessary information for the build, such as version number, release notes, and app description. Then select the appropriate groups or users to invite for testing. Finally, choose the desired release options, including automatic or manual release and availability for external testing.

Builds that go to users outside the organization will need to be reviewed by Apple. If an app includes specific features that require beta app review, such as Apple Pay, it may need additional review details as well. Provide any necessary information and wait for the review process to complete. For external testers, beta app review is required before distribution. Once everything looks good, click the "Submit for Review" button to initiate the review process.

Once an app is approved or released, TestFlight will send an email to the testers you invited. Testers can then download the TestFlight app from the App Store and access the app for testing.

Tip Remember to regularly monitor feedback from testers, address any reported issues promptly, and iterate on your app before submitting it for production release.

Please note that these instructions are based on the general process for posting an app to TestFlight, and there may be minor variations depending on updates to Apple's developer tools and processes. In fact, it's likely that Apple will change at least minor things about the process before this book hits the shelf! It's always a good idea to refer to the latest official documentation provided by Apple for the most accurate and up-to-date instructions.

Additional Compiling Options

There are a number of options in Xcode that can automate or help with various tasks when compiling software. Most require a separate service or script to be invoked and include the following:

- Build Scheme: Xcode build schemes allow developers to define custom configurations for building and running an app. Specify the target devices, build configurations (Debug/Release), and other options to exercise greater control over the build process and run different configurations of a given app.

- Command Line: Compile Xcode projects and workspaces using `xcodebuild` (and Swift packages with `swift build`). This provides flexibility and control over the build process but requires more knowledge of build tools and command-line syntax.
- Custom Build Scripts: Define custom build scripts that can be executed before or after the main build process. This allows automating tasks such as generating resources, running unit tests, or deploying an app to a server.

The best way to compile a Swift app depends on the specific needs of the process, which evolve over time. When starting out, building and running an app through Xcode is the easiest option. As a project grows and the requirements become more complex (especially when "compliance" becomes a common word in use), most will need CI/CD tools for greater control and automation.

CI/CD

CI/CD stands for continuous integration and continuous delivery. Essentially, it's a workflow that automates various stages of software development. This can mean faster deliveries, so no more waiting for lengthy manual deployments. CI/CD tools automate the delivery process, getting updates to users quicker. When done properly, there are also fewer bugs: with automated testing and integration happening early and often, bugs are caught and fixed before they reach users. Finally, as teams grow, developers can focus on writing code and innovating, instead of being bogged down with repetitive tasks.

The first part of CI/CD is continuous integration. That mostly refers to merging code between developers, automating builds, and automating tests. Continuous integration usually means that developers merge their code changes frequently, preventing conflicts and integration nightmares. Every code change then triggers an automated build process, ensuring compatibility and catching errors early. Finally, automated tests run after each build, verifying functionality and preventing regressions.

The second part of CI/CD is continuous delivery. This involves the delivery or deployment of the software. Deployment automation means that code changes are automatically deployed to different environments. This includes testing, staging, and production for further testing and feedback. That happens to give the ability to go back

in time (especially when apps are fully self-contained), which is commonly referred to as a "rollback." If something goes wrong in production, automated rollbacks minimize downtime and damage. Finally, there's telemetry in the process and resultant artifacts. That means monitoring and feedback can drive healthy application performance and feedback for future improvements.

For a concrete example, a CI/CD pipeline might run every time code is pushed to a repository, build the app, execute unit/UI tests, and then distribute a new beta build to a TestFlight group automatically.

There are tons of CI/CD tools, but the biggest include

- *Xcode Cloud*: Cloud-based service (with a free tier) from Apple for building, testing, and deploying Swift apps
- *Jenkins*: A free and open source platform for building and automating CI/CD pipelines
- *CircleCI*: A cloud-based CI/CD tool known for its ease of use and scalability
- *Travis CI*: A hosted CI service that has historically been popular for automating builds and tests
- *GitLab CI/CD*: Built-in CI/CD features within the GitLab platform, streamlining development workflows
- *AWS CodePipeline*: A cloud-based CI/CD service from Amazon Web Services

There are entire books and classes on each of these tools, so delving too deep into their bowels is a bit beyond the scope of this book. Having said that, Xcode Cloud is new (as of the time of this writing), and so it's worth looking at how to perform basic tasks with it.

Xcode Cloud

Xcode Cloud is one of the more accessible tools for newer developers and development teams without entrenched workflows based on other CI/CD tools. It's hosted in the cloud and there is a free tier with limited features, but paid plans cater to larger teams and more demanding projects.

Xcode Cloud takes care of a few things that other tools can't necessarily do. One of the things it helps fix is certificate distribution for developers. Xcode Cloud removes the need for more manual build triggers: code also gets automatically built whenever changes are pushed, ensuring the latest version is always tested and ready. That means an army of virtual devices in the cloud that can catch bugs before they reach users and potentially faster feedback when they do. Xcode Cloud, like any other CI/CD tool, is about collaboration. The devops tools keep everyone in the loop with detailed reports and seamless visibility into the development process.

Xcode Cloud does come with some requirements. First, it requires a supported version of Xcode (currently Xcode 15.0 or later). Second, projects need to be configured to use cloud builds and tests. Workflows then need to be built in Xcode Cloud that define builds, tests, and deployment steps. This can include testing apps in different stages, like development, staging, and production, to ensure smooth transitions. In addition to the expected unit tests, it can also be configured to do UI tests and performance tests. These can be done on specific branches to track progress and experiment with different features. Finally, everyone stays informed with real-time notifications and detailed reports on build status, test results, and deployment logs.

The Xcode Cloud documentation is the best place to get started, available at `https://developer.apple.com/documentation/xcode/xcode-cloud` for more information.

Use the Jenkins Xcode Plug-In

It's also possible to look at how to configure Jenkins real quick, as an example of how these work. To do so, first access the Jenkins dashboard and navigate to Manage Jenkins and then click Manage Plugins. Next, select the Available tab and search for Xcode integration. Install it and restart Jenkins.

Once installed, it's time to configure Jenkins. Create a new Jenkins project or job. Under Build, add a build step and choose the newly available Xcode option. Then fill in the required fields:

- Project/Workspace path: The path to your Xcode project or workspace
- Targets: The specific targets to build (e.g., "MyApp")
- Scheme: The build scheme to use (if applicable)
- Configuration: The build configuration (e.g., "Debug" or "Release")

If the project requires code signing or provisioning profiles, store the necessary credentials in Jenkins. To add them, go to Manage Jenkins ↗ Manage Credentials. Then add credentials for the Apple Developer account and provisioning profiles. To access signing certificates on the build machine, unlock the macOS keychain if needed. Then add a build step in Jenkins to unlock the keychain using the security command.

Once the job is built, run the Build. Save the job configuration and trigger a build manually or automatically based on triggers that were set up. Jenkins will then execute the Xcode build process and provide build logs and results.

Many CI/CD tools can also be integrated with tools like JFrog's Artifactory or Swift Registry for artifact management. There are also plenty of well-documented workflows and code to get started with on GitHub and GitLab. Most tasks, though, like the one above, require developers to have access to an API key if the task will perform any final tasks beyond just compiling (e.g., submitting or uploading an app).

Getting an App Store Connect API Key

Apple's App Store Connect is where developers upload software to get it into one of the distribution channels so it shows up on the App Store. This might be to upload apps directly to the App Store for review and publishing or to TestFlight once we've managed to archive an app in Xcode. Manually moving apps has always been a pain and the keys that people need access to are a bit much. We can now create workflows in Xcode that hook directly into other distribution technologies, like Jenkins. Those workflows often require an API key.

In App Store Connect, the Account Holder requests API access once. After access is enabled, team API keys can be generated by an Account Holder or Admin. To do so, open App Store Connect, click Users and Access, then Integrations, and click Request Access if it has not already been granted like in Figure 1-4.

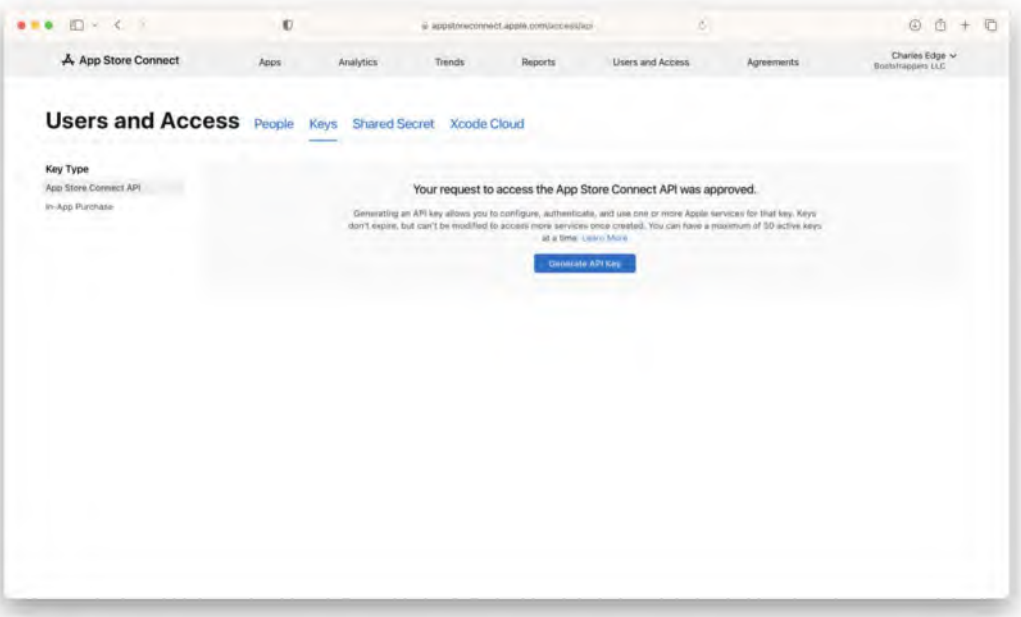

Figure 1-4. *Set up an API key in App Store Connect*

From the Keys screen, once the ability to use the API has been granted (Figure 1-5), click Generate API Key.

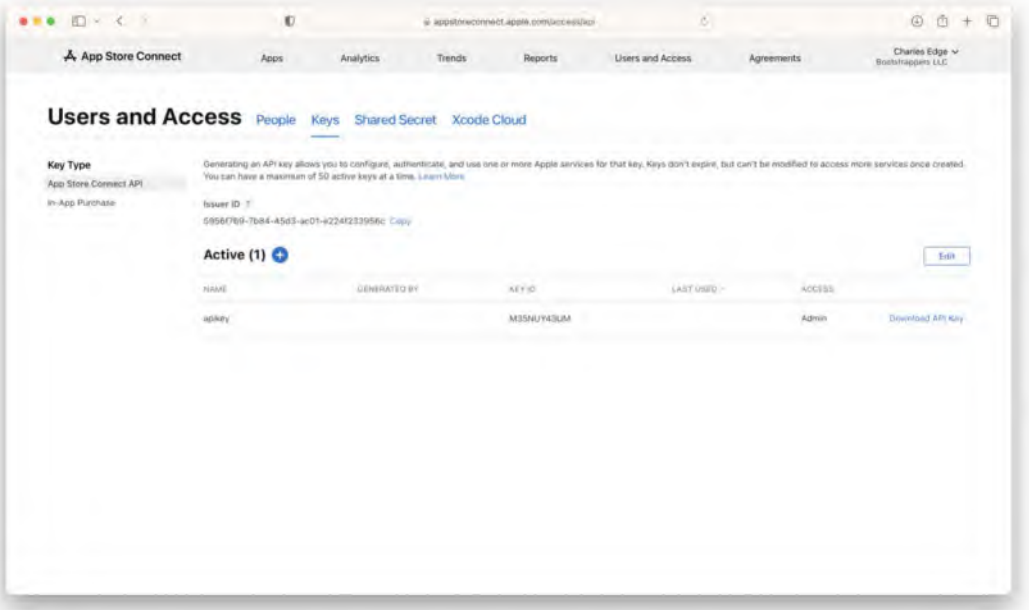

Figure 1-5. *API keys in App Store Connect*

At the list of keys (which will initially be empty), click the plus sign to create a key. At the Generate API Key dialog, Figure 1-6, enter a name for the key and choose the key's access level/role.

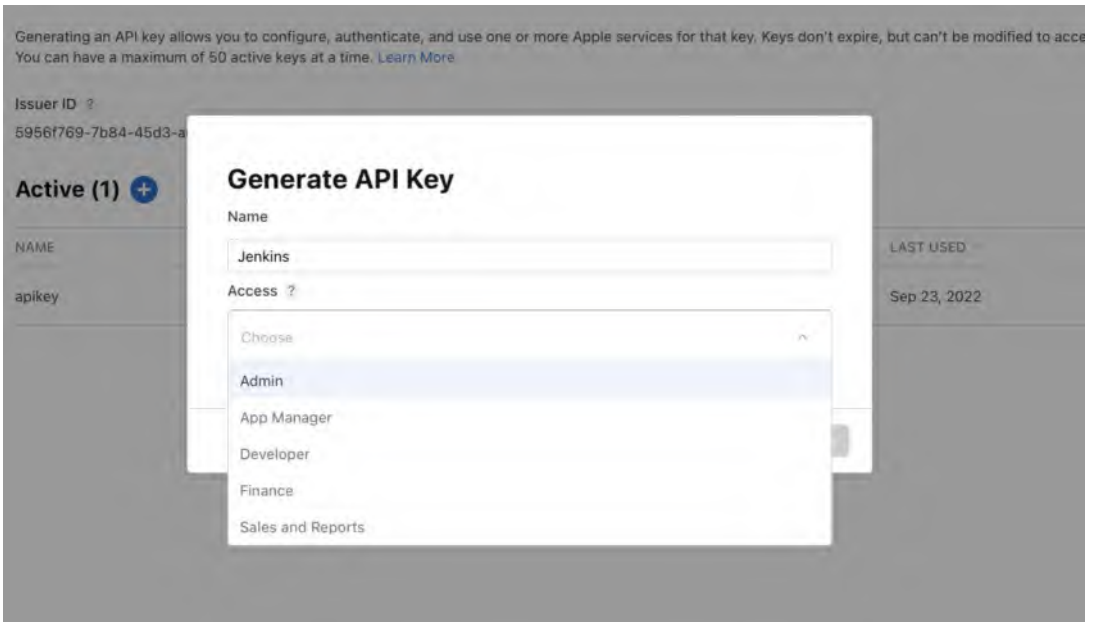

Figure 1-6. *Generate an API key*

When the key is created, download the `.p8` file immediately, Figure 1-7, and store it securely. Apple only allows that private key file to be downloaded once.

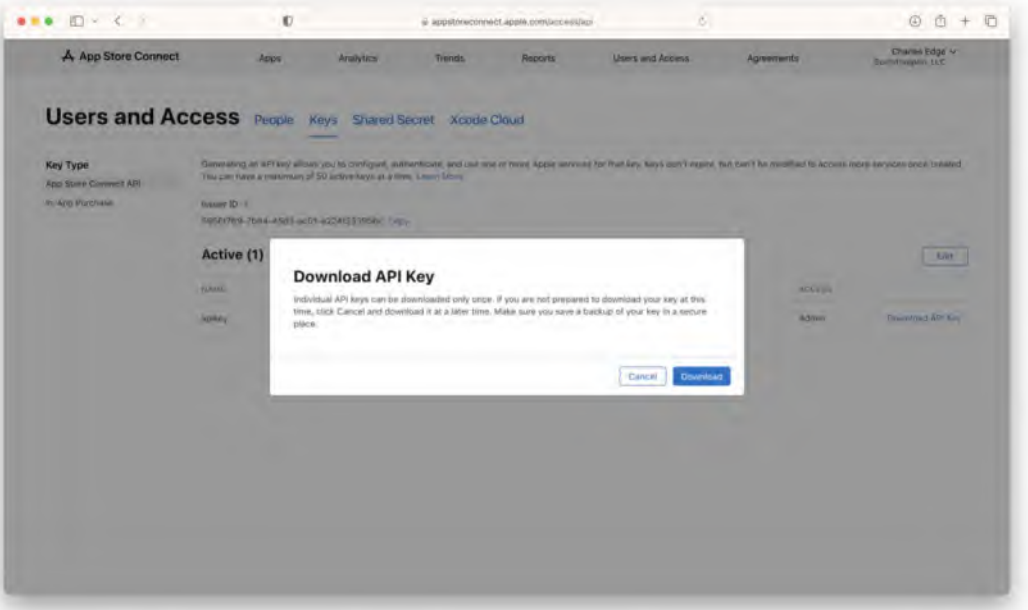

Figure 1-7. *Download an API key*

The API is described at `https://developer.apple.com/documentation/appstoreconnectapi`. JWT tokens created with these keys use ES256 signatures and include key metadata such as the key ID. The API actions available to a token are determined by the role/permissions associated with the key.

The API covers many App Store Connect workflows, including build metadata, testers, app versioning, pricing, and localization. This supports in-house workflows where developers can request various actions, like a self-service portal, rather than having to have an admin do everything manually.

Once users have access to apps in TestFlight and apps are getting closer and closer to being done, it's time to start to think about a mass release, if that's the intent. That means it's time to deal with the App Store submission process, which some developers think is simple and others think is draconian and evil and horrible. The truth is really just a matter of perspective. Apple tries to protect users and can overindex on that in defense of rules that might not be entirely necessary. Some developers think it's the era of BASIC programming on an Apple II and that they should be able to read and write directly from memory. The hacker movement moved from hardware and into the sandbox Apple provides long, long ago.

Introduction to the App Store Submission Process

The Apple App Store submission process is the process of submitting your iOS, iPadOS, macOS, tvOS, watchOS, or visionOS apps to the App Store for review and publication. The mechanics of a submission are fairly simple. To submit an app to the App Store, the following are required:

- Enroll in the Apple Developer Program and ensure App Store Connect access. This provides the portal where apps and submissions are managed.
- Register the app with Apple. This will create a unique identifier for the app and automatically create the entry to start creating a product page for it.
- Create a product page for your app on App Store Connect. This is where metadata is provided about an app, such as its name, description, screenshots, and preview videos. It's also where things like encryption export requirements are set.
- Upload the app build (archive) and any required App Store metadata assets, such as screenshots and previews.
- Provide information about your app, such as its name, description, and keywords.
- Submit your app for review. Once you have submitted your app, it will be reviewed by Apple. If your app is approved, it will be published on the App Store.

Distribution of Signed Apps

There are plenty of apps that aren't distributed through the Apple App Store. This can be done for a variety of reasons, like the app can't be properly sandboxed, the app is meant for internal use at a company, etc. But it requires a lot of very specific planning. Given that systems administrators are often creating tools for use internally and not on an App Store, it's worth digging into this – namely, with open source apps.

Using Open Source Apps

There are plenty of apps out there that can be beneficial to an organization but don't really make sense to live on an App Store. This might be because the app uses private APIs, breaks acceptable design patterns, needs to be customized for every use case, is just proof of concept code, etc. Anyone with the source code and toolchain can usually compile for local testing; distribution signing requirements are where certificates and provisioning become critical. We can go into more detail later for people that want to then distribute/redistribute those apps.

To get started, first we'll clone the project to our local machine. To do so, in GitHub or GitLab or wherever it lives, from a machine with Xcode, let's just click the Clone button, and select Xcode (using NoMAD as an example here in Figure 1-8).

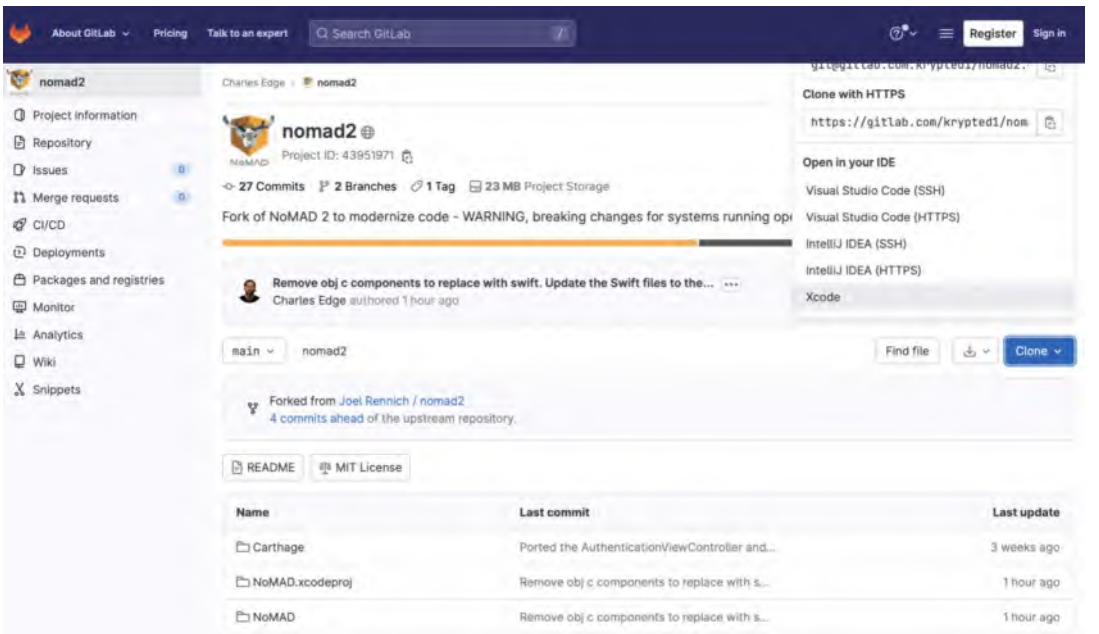

Figure 1-8. *The NoMAD2 open source project in GitLab*

The deep link opens Xcode (or asks to do so), and then you have an option for which branch you want to clone. There may be multiple branches if there are different developers or epics or features being worked on, but for the purposes of this book, we'll just use "main". Click Clone, Figure 1-9, select a location, and the project will download to the computer.

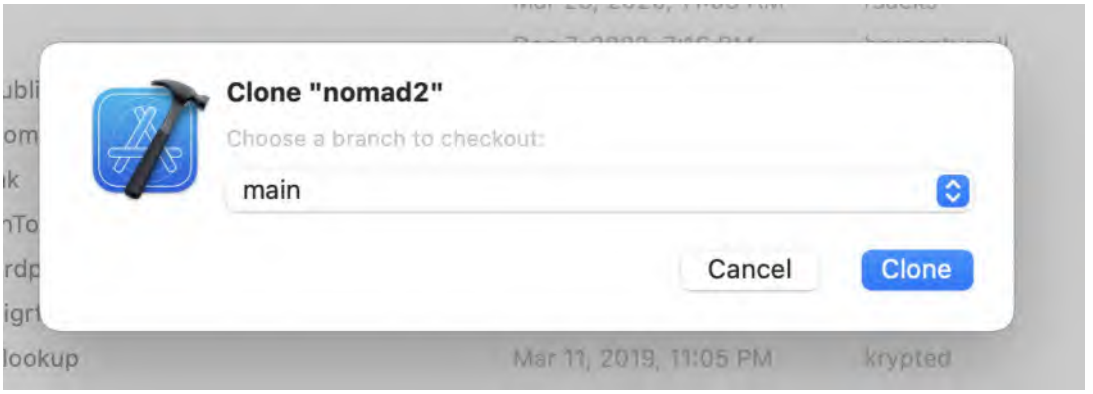

Figure 1-9. *Cloning a repository in Xcode*

Once downloaded, open the `.xcodeproj` file (or `.xcworkspace` if the project uses one) in the root of the directory it was downloaded to. You should see a project structure like Figure 1-10.

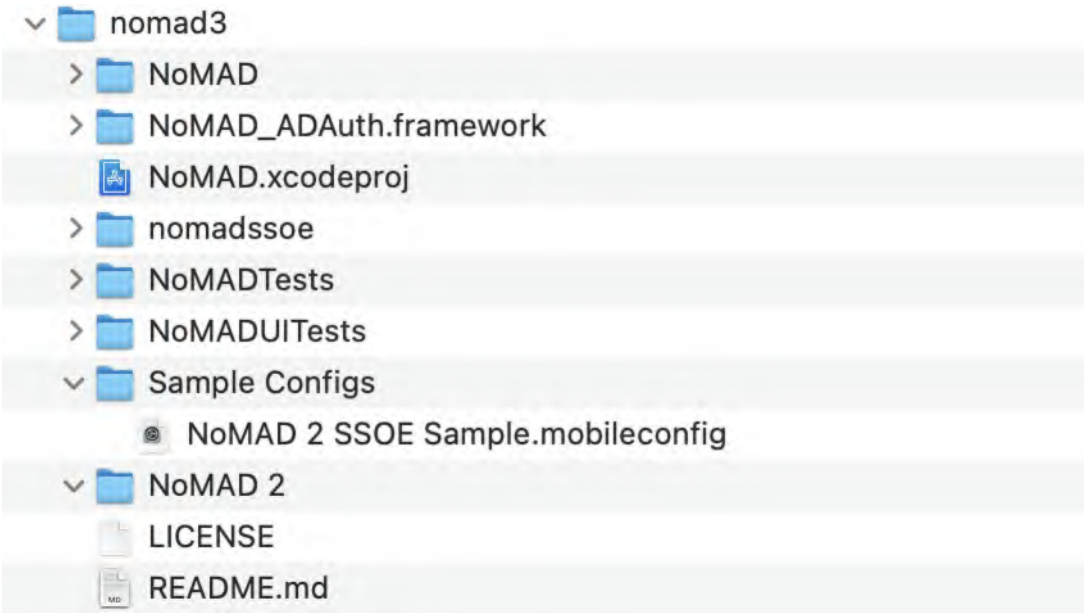

Figure 1-10. *An Xcode project structure*

At this point, if the project uses Swift packages, we might need to wait for those to download fully to compile. Under Package Dependencies, check to see if those have a status that they're downloading. In general, if it's not greyed out a little and all the files

show, you're fine. It usually takes less time than it'll take to do the next steps, so don't worry about that unless you get an error later.

Next, let's choose our developer certificate. We need to do this for each "target" we'll be compiling for. Click on the name of the project in the file tree and then Signing & Capabilities in the tab along the top of the screen.

Once each target has been appropriately provided with the correct signing certificate, click the build button (looks like a play button) toward the top of the screen. The app will compile, provided there are no errors. Note that in the screen below, we see warnings, but they're yellow, not red, so all good. Head back to the Finder and the app should be open (in this case, it's a menu bar item so won't appear in the Dock). To then see the compiled app, click the Product menu and select Show Build Folder in Finder.

Nested in that Build folder is the product you seek. Whether it runs on another machine depends on how it was signed and distributed. For more on how to make it more widely distributable, see https://developer.apple.com/documentation/xcode/preparing-your-app-for-distribution.

Resolving Signing and Entitlement Issues

Many of the tools found for MacAdmins are available in both raw source code and in the form of a compiled binary. For those who didn't want to use the precompiled application bundle, there is often some tweaking that will need to be done to get it to work. Many software packages have permissions to do various tasks.

NoMAD is a tool most MacAdmins will be well aware of. NoMAD interacts with the keychain, so it will have to use a TeamID, or, to expand the term, the Team Identifier Prefix. This means the new version won't be able to access keychain items created by previous versions of NoMAD, which use the creator's prefix (we didn't reference Joel as "THE Creator" – but "a creator" to be clear). Ergo, this article is really just for helping those who want to test versions as they're being built or do net-new deployments. A local script could theoretically be used to augment/migrate keychain items, but that seems like a total pain, especially when considering that future versions of NoMAD will have to retire deprecated APIs and so will need to choose to cease functioning for a given version of macOS, and it's easy in most deployment tools to scope old NoMAD to a specific version of the OS and a newer version, perhaps with your own TeamID, to newer versions of the OS. But deployment options are beyond the scope of this article.

To find the TeamID, from Xcode, click on the main project and then Signing & Capabilities, as seen in Figure 1-11, – voilà, there it is.

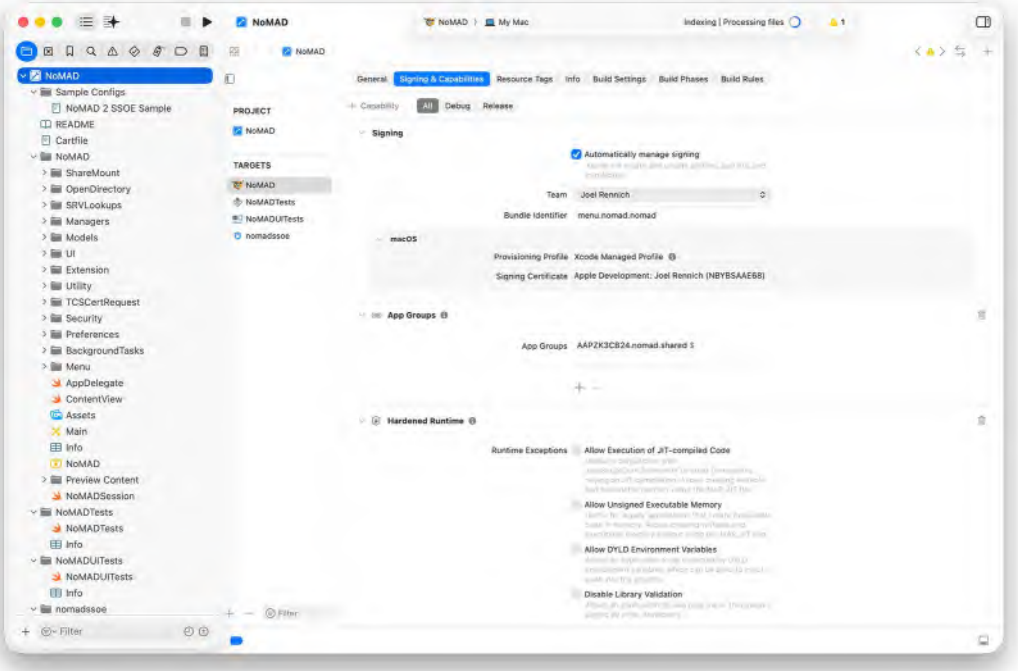

Figure 1-11. *Signing & Capabilities section of the project in Xcode*

Now let's search for the string of the old TeamID (in this case, "VRPY9KHGX6") in the Project Navigator pane (using the magnifying glass above the file list on the left of the screen) as showing in Figure 1-12.

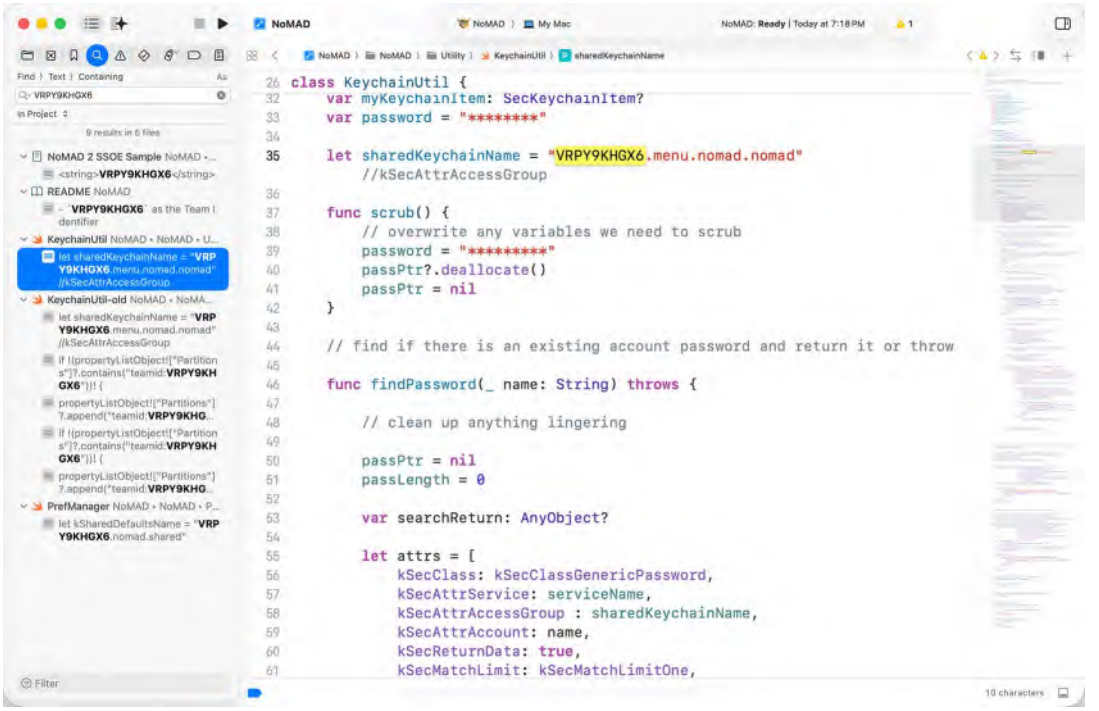

Figure 1-12. *Finding Team Identifiers in Xcode*

Each of these needs to be changed (notice in the search results there are a few different entries). Once done, if using MDM to approve the entitlement, the "Team Identifier" referenced in the README would be different as well, so this app bundle would deploy with a different policy (Figure 1-13).

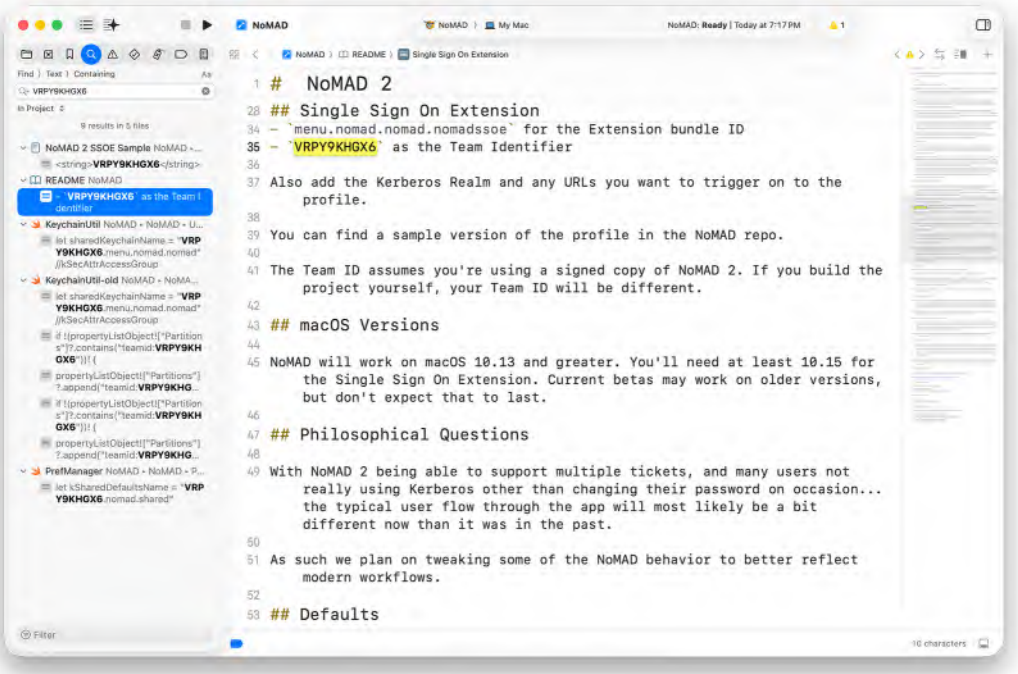

Figure 1-13. *Editing Team Identifiers in Xcode*

This chapter has up to this point primarily dealt with getting software distributed. There's one other aspect to cover. Much of the book itself has dealt with writing code, except the section on SwiftUI. It's worth spending a little time covering how to link those two aspects together.

Connecting User Interface Elements to Code

To connect SwiftUI user interface elements to Swift code, use the @State and @Binding properties. The @State property is used to store the state of a user interface element, and the @Binding property is used to bind a user interface element to a state property.

To bind a user interface element to a state property, pass a `Binding` (e.g., `$text`) to controls that accept bindings.

For example, the following code shows a text field that is bound to a @State property:

```
@State var text = ""
TextField("Enter some text", text: $text)
```

The $text expression tells SwiftUI to bind the text field to the text state property.

When the user changes the text in the text field, the text state property will be updated. Use the text state property in code to do whatever is necessary with the text, such as displaying it in a label or saving it to a file. Also use the @Binding property to bind multiple user interface elements to the same state property. For example, the following code shows two text fields that are bound to the same @State property:

```
@State var text = ""
TextField("Enter some text", text: $text)
TextField("Enter some more text", text: $text)
```

When the user changes the text in either text field, the text state property will be updated. These are fairly basic but will get anyone started linking different elements.

Conclusion

Writing code and shipping code are two different disciplines. Tthis chapter was about the far less glamorous work of getting your app onto someone else's device. That path starts in the Product menu, where Build and Run, Test, Profile, Analyze, and Archive each produce a different kind of artifact, and it runs through targets, which are how a single project turns into a free version and a paid version, an app and its extensions, or builds for several operating system versions at once. Along the way there's the simulator for the platforms that don't run natively on a Mac, and `simctl` for the moments when clicking through the simulator by hand stops scaling.

Underneath all of it is code signing, which is the invisible hand that decides whether a build runs on one machine or a million. Certificates, entitlements, provisioning, and notarization aren't obstacles placed in the way of developers so much as the mechanism by which the operating system decides to trust a binary at all - and most of the frustrating build errors in this chapter trace back to a mismatch somewhere in that chain. Once signing is sorted, the distribution options line up along the friction spectrum described at the start of the chapter: TestFlight for internal and external testers, Ad Hoc and Enterprise builds for managed devices, the App Store for everyone else, and directly compiled open source projects for the tools that were never meant for a store in the first place. Re-signing something like NoMAD with a different TeamID is a good illustration of how deeply that identity is woven into an app - it isn't just a certificate, it's what the keychain and MDM policies key off of.

The rest of the chapter was about removing manual steps. App Store Connect API keys, Xcode Cloud, Jenkins, and the other CI/CD tools all exist because building, testing, signing, and uploading by hand is exactly the kind of repetitive work that invites mistakes. Start with the simple path - build and run in Xcode, archive, upload to TestFlight - and automate the parts that hurt as the project grows. Apple will change the details of these workflows, likely before this book has been on a shelf very long, but the shape of the process tends to stay the same. And with the brief look at wiring SwiftUI elements to code through @State and @Binding, the writing side and the shipping side of this book finally meet in the middle.

The next chapter covers errors, debugging, and instrumentation.

CHAPTER 2

Errors, Debugging, and Instrumentation

Figuring out what different errors mean is one of the hardest parts of writing code. It's hard to say how what exactly has been "thrown" when an error is, well, thrown. Sure would be nice if IDEs like Xcode could automagically tell developers what is wrong; sometimes they do, and Xcode can even make corrections on behalf of developers. But a lot can go wrong, and that's not something that always happens. Thus, much cursing and banging a head on a keyboard can be necessary. If there were a simple way to explain how to always fix errors, then there would be little need to write a chapter on it. Might be a better tweet at that point. But alas, that's what separates those who have the patience to be a software developer from those who don't.

It's worth pointing out that most errors come in one of two forms: compiler errors and errors that crop up after an app compiles. This can include things like apps crashing when certain buttons are hit or apps crashing when left open too long (which shouldn't be a thing but can happen when memory isn't being managed properly). Compiler errors are usually immediate in Xcode and can often be searched for online. When apps are performing badly and/or crashing, there might also be errors in logs. There are also instrumentation logs, build logs, and errors when analyzing software.

"Let Me Google That for You"

Even the most seasoned developer encounters errors in their code. Some are compile-time, some far more challenging to isolate. It is an inevitable part of the journey. It's never been easier to find the source of errors than it is today. There are sites dedicated to developers helping one another, forums by the companies that make languages (including Apple), official documentation, and a veritable wealth of resources available

C. Edge et al., *Beyond Scripting*, https://doi.org/10.1007/979-8-8688-3035-8_2

to help decode those cryptic messages and get code back on track. The journey should probably begin with Google. But that can net a very wide array of responses and so searching in specific locations can be a great way to get more direct information.

The first place to check is the Apple Developer Forums. Direct from the source, this is a place to get expert advice and community insights straight from Apple's own forums. Developers can explore discussions on a wide range of Swift and Xcode issues, from specific errors to best practices. There are also potential fixes and workarounds shared by Apple engineers and experienced developers, including the developers who built the specific APIs that are producing errors in many cases, as shown in Figure 2-1.

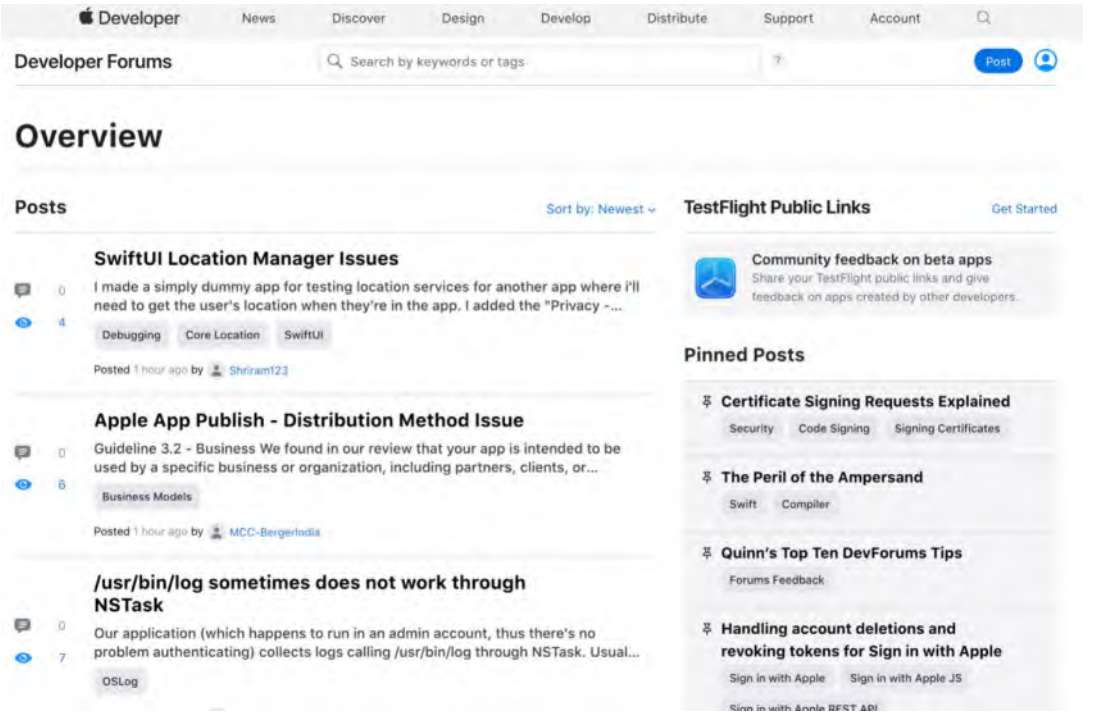

Figure 2-1. *The Apple Developer forums*

Stack Overflow did as much to democratize programming as anything else since Grace Hopper forced programming out of the hands of mathematicians in the hallowed halls of mainframe programmers. Stack Overflow, Figure 2-2, has become a vast knowledge base and anyone can tap into a massive repository of questions and answers from developers around the world. Responses to questions include detailed

explanations of problems that other developers have encountered, and anyone can benefit from clear explanations, code examples, and alternative approaches to problem-solving, sometimes within minutes of posting questions.

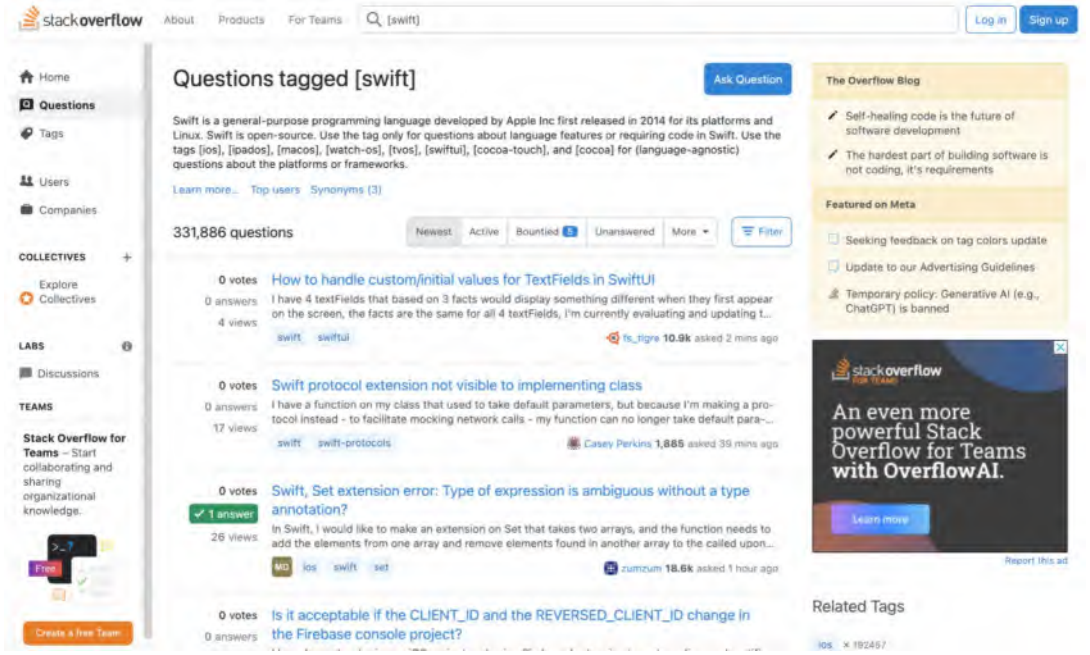

Figure 2-2. *Stack Overflow*

Another place where developers can find a massive trove of information is the swift.org forums. This is the official Swift community where developers can engage in discussions and problem-solving. It's got a language-specific focus where developers can get help with errors related to the Swift language itself, as well as Xcode integrations. Developers can also stay updated with the latest Swift developments and best practices.

There are also a number of sites dedicated to learning Swift. Hacking with Swift, Codecademy, and Swift courses from Udemy are all great places to hone coding practices and find areas for improvement. The comprehensive tutorials allow developers to learn Swift and Xcode fundamentals through well-structured tutorials and courses. There's also error-specific guidance, like articles and resources dedicated to common Xcode errors and their solutions, as well as a supportive community of learners and developers for assistance and discussions.

Bloggers often get surfaced in Google searches for errors. Some, like Ray Wenderlich, also develop high-quality tutorials that help developers explore in-depth tutorials and articles covering various Swift and development topics. There are troubleshooting sections available at sites like raywenderlich.com that help developers with dedicated sections addressing common Xcode errors and debugging techniques.

Xcode now allows for you to connect to various AI services. These can also be very helpful in trying to get to the bottom of errors. However, care should be taken to ensure that more errors aren't being introduced by the AI at the same time.

Xcode also comes with a built-in help option. Developers can access documentation and explanations for errors directly within Xcode. Sometimes, there's a button to just fix issues like in Figure 2-3. The help dialog in Xcode allows developers to refine search terms to access specific error messages and keywords and so find relevant solutions online. That can include clear code snippets. But the most important thing when trying to find help is to practice patience and persistence. Troubleshooting can be a process of trial and error; don't give up! Errors are opportunities to deepen understanding.

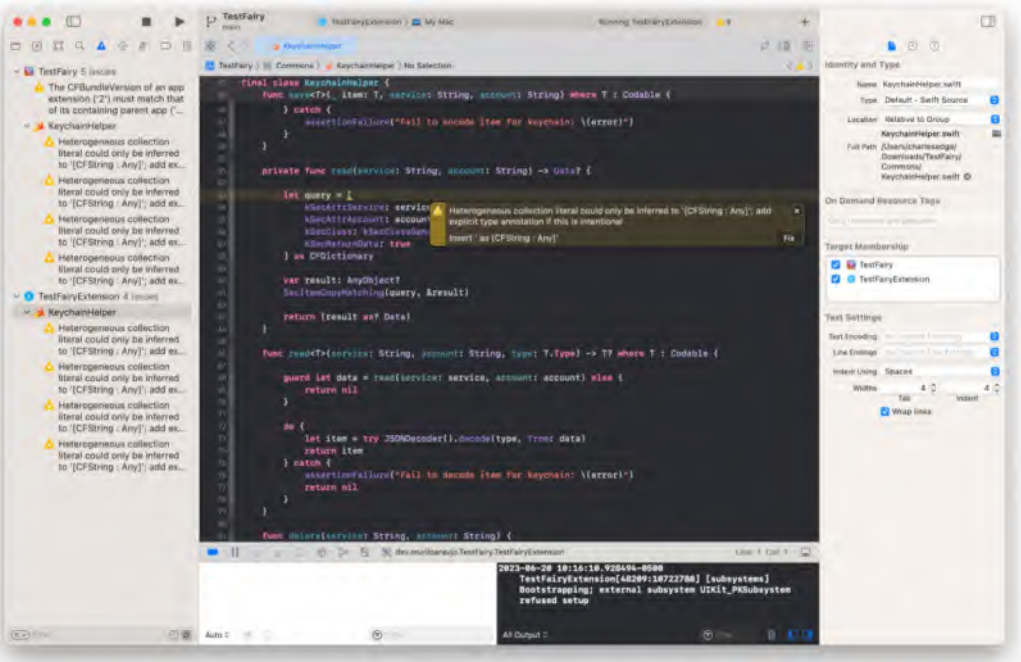

Figure 2-3. *The Xcode editor*

Again, just Googling errors (in quotes) will find a lot of tips and tricks to fix problems that come up. But finding the exact error to search for, if they don't show up as a yellow or red error in Xcode, can be a challenge. External resources help narrow the possibilities; the next step is applying that guidance directly while debugging in Xcode.

Navigating Xcode's Error Messages

Encountering errors in Xcode is a natural part of the development journey. The key to overcoming them lies in understanding their nuances and implementing effective debugging strategies. A useful skill is just recognizing error types, which include the following:

- *Compiler errors*: These red flags indicate issues with code syntax, structure, or type mismatches. Address them before building projects.
- *Build errors*: Occur during the build process, often involving missing resources, incorrect configuration, or dependency conflicts.
- *Runtime errors*: Emerge while the app is running, such as crashes due to invalid operations or memory issues.

Interpreting the error messages is another skill altogether. It's easy to forget to actually read dialogs. But read error messages in Xcode carefully for hints about the problem's nature and location. Also look for (or add) file names and line numbers in error messages. That's a solid way to pinpoint the exact code section causing an error. Finally, the Issue Navigator provides a list of errors with detailed descriptions.

Some common error types and solutions include the following:

- *Syntax errors*: Includes missing braces, parentheses, commas, or quotes; incorrect variable or function names; and malformed expressions
- *Type mismatch errors*: Includes attempting operations on incompatible data types
- *Initialization errors*: Includes using variables before assignment

- *Linker errors*: Includes missing framework or library references and incorrect file paths, or misconfigured build settings
- *Runtime errors*: Includes things like array index out of bounds and forced unwrapping of optionals with nil values

There are some strategies to debug these errors. Breakpoints are used to pause execution at specific lines for inspection. Print statements are used to display variable values for tracking code behavior. Step-by-step debugging executes code line-by-line to pinpoint errors. Using console output allows developers to view messages and logs for clues. Additionally, it can help to do a clean build. This means clearing cached data and rebuilding the project for a fresh start.

Compiler Errors

Compiler errors are the first line of defense against coding mistakes that prevent Xcode from building an app until they're addressed. These are identified, as noted previously, with red underlines and markers. Xcode visually highlights code lines with compiler errors. The Issue Navigator provides a detailed list of errors with descriptions and locations.

Common compiler error types include the following:

- *Syntax errors*: Includes missing braces or parentheses, incorrect uses of keywords, and misspelled variable or function names
- *Type mismatch errors*: Includes assigning incompatible data types
- *Initialization errors*: Includes using variables before initialization
- *Variable scope errors*: Includes accessing variables outside their defined scope
- *Circular references*: When two entities strongly reference each other, creating a memory leak by preventing deallocation
- *Missing required members*: Failing to implement mandatory methods or properties in classes or protocols

Once errors are classified, they can be more easily searched for and their root causes triangulated to resolve them holistically. Also review surrounding code. Consider declaring variables within the appropriate scopes, restructuring code to eliminate

dependencies, and making sure the required methods or properties are provided. Use Xcode's real-time editor diagnostics to identify errors while typing and Fix-it suggestions to offer potential solutions for common errors. Just remember that compiler errors are allies when writing correct and maintainable code and to be systematic when approaching debugging and error resolution. Next, let's look at how to use instrumentation options in Xcode.

Instruments in Xcode

Instruments is a tool that can be used to debug Swift code from Xcode. It allows you to collect data about your app's execution, such as CPU usage, memory usage, and network traffic. This data can be used to identify performance bottlenecks and other problems in your code. To use Instruments, follow these steps:

1. Open your Xcode project.
2. In Xcode, choose Product ➤ Profile.
3. In Instruments, select the appropriate template and click Record.
4. Run the app.
5. Once the app has finished running, view the results in Instruments.

The Instruments window shows a variety of data about your app's execution, most notably:

- *CPU usage*: This shows how much of the CPU an app is using.
- *Memory usage*: This shows how much memory an app is using.
- *Network traffic*: This shows how much network traffic an app is generating.

Use this data to identify performance bottlenecks and other problems in your code. For example, if an app is using a lot of CPU resources, use the Instruments data to identify the specific lines of code that are causing the high CPU usage. Once problems are identified, fix the code and then rerun the app. Then use the Instruments data to verify that the problem has been fixed.

There are other options within Instruments. Use the Filters bar to filter the data. This can help focus on the data that is most relevant to problems. Use the Timeline view to see a graphical representation of an app's execution. This can help identify the specific lines of code that are causing performance problems. Use the Call Tree view to see a tree-like representation of an app's call stack. This can help identify the functions that are calling each other. Another place to look for errors is in logs.

Reviewing Logs

Apple has a number of different logging APIs. For the past few releases, devices can capture everything possible in logs, creating what many administrators and developers might consider to be a lot of chatter. As such, an entirely new interface needed to be developed to categorize and filter messages sent into system logs.

Writing Logs

The logger command is used to create entries in system logs from the command line. When using tail to view /var/log/system.log, notice that entries aren't written on the fly. This is because logs created in macOS are more complex than those in most Unix-based derivatives.

Let's take a simple log entry. Below, the string "Hello Logs" is written into the system log. To do so, use the –i option to include the process ID of the logger process and –s to write to the system log, as well as to stderr. To make the entry easier to identify, tag it with –t followed by the tag string. And finally, quote the entry written into the log. This is basically the simplest form of an entry:

```
logger -is -t krypted "Hello Logs"
```

Once written, use the log command to read the spiffy new entries. This isn't terribly different than how things worked in earlier versions of the Mac. Developers should note that legacy APIs such as `asl_log_message` and `syslog` are deprecated, and `NSLog` is still supported but is not preferred for new structured logging. Unified Logging is available on macOS 10.12+, iOS 10+, tvOS 10+, and watchOS 3+, and modern Swift code typically uses `Logger` from OSLog (or lower-level `os_log` APIs).

Reading Logs

Logs are then stored in tracev3-formatted files in /var/db/diagnostics, which use a compressed binary format. As with all binary files, you'll need new tools to read them. The Console app has been updated with a new hierarchical capability and the ability to watch activities, subsystems, etc.

The log command provides another means of reading logs. To get started, first check out the man page:

```
man log
```

That "Hello Logs" string used earlier is part of a message used to view entries easily with the "log show" command. The example below runs a scan of the last three minutes, using the –last option and then providing a –predicate. Those are explained a bit later, but think of them as query parameters. Here, simply specify to look for "Hello Logs" in eventMessage:

```
log show --predicate 'eventMessage contains "Hello Logs"' --last 3m
```

Filtering the log data using `eventMessage CONTAINS "Hello Logs"` shows that entries appear as follows:

```
Timestamp                       Thread     Type        Activity             PID
2023-03-23 23:51:05.236542-0500
0x4b83bb   Default     0x0                          88294  logger: Hello Logs
-------------------------------------------------------------------------------

Log   - Default:    1, Info:        0, Debug:     0, Error:    0, Fault:  0
Activity - Create:       0, Transition:           0, Actions:             0
```

Next is what to use where. Here's an example to find all invalid login attempts. Many will prefer the "log stream" command. It's possible to just use show again, because it looks better and is easier to read. Use log with the syslog –style so it's easier to read, and then just look at everything connected by specifying a space instead of an actual search term:

```
log show --style syslog --predicate 'eventMessage contains " "' --info
--last 24h
```

Looking at the output shows an entry similar to the following:

```
2023-03-23 14:01:43.953929-0500  localhost authorizationhost[82865]: Failed
to authenticate user <admin> (error: 9).
```

Next, search for "Failed to authenticate user" to count invalid login attempts. This can be placed into a command or other app that, for example, could build a Jamf extension attribute. Then just find each entry in eventMessage that contains the string, as follows:

```
log show --style syslog --predicate 'eventMessage contains "Failed to
authenticate user"' --info --last 1d
```

As with many tools, once a couple of basic incantations are understood, the ecosystem becomes infinitely simpler to understand. The most basic incantation of reviewing logs is just "log stream" without bothering to constrain the output:

```
log stream
```

Running this is going to spew so much data into a terminal session, though. So to narrow down to specific searches, let's look at events for an identifier, like com.krypted:

```
log stream --predicate 'eventMessage contains "com.krypted"'
```

View other logs and archives, by calling a file name:

```
log show system_logs.logarchive
```

Logs are then organized and classified in specific ways.

Organization and Classification

The logging format also comes with subsystems. Developers can file messages into, for example, a com.yourname.whatevers domain space to easily find log messages. Also build categories and, of course, as noted previously, tag messages. So, there are about as many ways to find log entries as can be asked for. Apple has a number of subsystems built into macOS. A list of Apple subsystems is in a class that you should be able to throw into Python and Swift projects at https://gist.github.com/krypted/495e48a995b2c08d25dc4f67358d1983.

There are also different logging levels. These include the basic levels of Default, Info, and Debug. There are also two special levels available: Fault and Error. All of this is to add hierarchical logs (which makes tracing events a much more lovely experience) and protect the privacy of end users (think sandbox for logs). Consider watching the WWDC session where Unified Logging was introduced, at https://developer.apple.com/videos/play/wwdc2016/721.

Another aspect worth mentioning for the MacAdmins out there is how to go about switching between logging levels for each subsystem. This is done with the "log config" command. Here, use the –mode option to set the level to debug, and then define the subsystem to do so with the aptly named –subsystem option:

```
log config --mode "level:debug" --subsystem com.krypted
```

For particularly dastardly apps, the above might just help troubleshoot a bit. As mentioned earlier, these predicates can be considered metadata searches. These include the following:

- category: Category of a log entry.
- eventMessage: Searches the activity or message.
- eventType: Type of events that created the entry (e.g., logEvent, traceEvent).
- messageType: Type or level of a log entry.
- processImagePath: Name of the process that logged the event.
- senderImagePath: Not all entries are created by processes, so this also includes libraries and executables.
- subsystem: The name of the subsystem that logged an event.

To make things just a tad bit more complicated, it's also possible to string together search parameters. There are a number of operators available to do so, similar to what's seen in SQL. These include

- && or AND to indicate two matches
- || or OR indicates one of the patterns matches
- ! or NOT searches for items that the patterns don't match for, which is useful for filtering out false positives in scripts

- = to indicate that one search matches a pattern or is equal to
- != to indicate that the search is not equal to
- > is greater than
- < is less than
- >= means greater than or equal to
- <= means less than or equal to
- CONTAINS indicates a string matches a given pattern with case sensitivity
- CONTAINS[c] indicates a string matches a given pattern without case sensitivity
- BEGINSWITH indicates a string begins with a given pattern
- ENDSWITH indicates that a string ends with a given pattern
- LIKE indicates a pattern is similar to what you're searching for
- MATCHES indicates that two text strings match
- ANY, SOME, NONE, IN are used for pattern matching in arrays
- NULL indicates a NULL response (e.g., you see "with error (NULL)" in logs a lot)

To put these into context, let's use one in an example. The most common use has been a compound search, matching both patterns. Here, let's troubleshoot the Multipeer Connectivity framework by looking at the WirelessProximity subsystem for Bluetooth. It's possible to see how often it's scanning for new devices, keeping both patterns to match inside their own parentheses, with all patterns stored inside single quotes, as follows:

```
log show --style syslog --predicate '(subsystem == "com.apple.bluetooth.
WirelessProximity") && (eventMessage CONTAINS[c] "scanning")' --info
--last 1h
```

Developers and systems administrators will find the Apple guide on predicate programming, available at https://developer.apple.com/library/prerelease/content/documentation/Cocoa/Conceptual/Predicates/AdditionalChapters/Introduction.html, to be pretty useful if doing lots of this kind of work.

Note sysdiagnose, a tool long used for capturing diagnostic information to include in bug reports, is still functional and now includes Unified Logging information, so Apple developers can get a complete picture of what's going on in systems.

Reviewing TCC Dialog Prompts

The log command provides a number of options to see various events on a Mac. Let's say an app was automatically denying a privacy permission prompt. First, let's find all prompts. Do so using the com.apple.TCC subsystem as a predicate. The command below simply pipes the output to grep for Prompting:

```
/usr/bin/log show --style syslog --predicate 'subsystem == "com.apple.TCC"'
--info --last 12h | grep Prompting
```

It's better to use "&& contains" in syslog because it might be more efficient, but grep is an option as well. Armed with the output, let's swap Prompting in the command above to deny and shorten the window for how long it takes to compile and run the app (typically less than an hour):

```
/usr/bin/log show --style syslog --predicate 'subsystem == "com.apple.TCC"'
--info --last 1h | grep Deny
```

For reporting purposes, this could also be used to generate a list of apps that have a binary_path to see which software users are granting permissions to:

```
/usr/bin/log show --predicate 'subsystem == "com.apple.TCC"' --info | grep
Allow | grep binary_path
```

There are plenty of other options to look at logs, both from the command line and the Console app. Thus far, this chapter has focused on reactively fixing errors. Now, let's turn the page toward a proactive approach.

Proactively Fixing Bugs with Unit Tests

Unit tests are automated tests that verify the functionality of a unit of code, such as a function or class. Unit tests are important because they help ensure that code is working as expected. For teams and admins maintaining apps over time, a reliable test suite makes refactoring, onboarding, and scaling releases safer and more predictable. When possible, it's best to plan code based on the idea of test-driven development, which can mean building tests before code, or at least writing code in such a way that it's testable.

To write unit tests, use the XCTest framework. The XCTest framework provides a number of classes and methods that can be used to write and run unit tests. To start writing unit tests, first create a test class. A test class is a subclass of the XCTestCase class. The test class should contain one or more test methods. A test method is a method that starts with the `test` prefix. Each test method should test a single unit of code.

Inside a test method, use the XCTest framework to verify the behavior of the code being tested. For example, use the XCTAssertTrue() method to verify that a condition is true, or the XCTAssertNil() method to verify that a value is nil. The following example shows a simple unit test for a function that calculates the sum of two numbers:

```
import XCTest
class SumTests: XCTestCase {
  func testSum() {
    let a = 1
    let b = 2
    let sum = sum(a, b)
    XCTAssertEqual(sum, 3)
  }
}
```

This test method verifies that the sum() function returns the correct result when the two arguments are 1 and 2. To run unit tests, use the Xcode test navigator. The test navigator is located in the upper-left corner of the Xcode window. To run all of the unit tests in a project, click the "Run All Tests" button in the test navigator. Or run individual unit tests by clicking on the test name.

Unit testing is an important part of the development process. By writing unit tests, you can help ensure that your code is working as expected. If you are developing a Swift program, I encourage you to write unit tests for your code.

When writing unit tests, it is important to test all of the possible inputs to the code. This will help ensure that code is robust and can handle unexpected inputs. Also test the different error conditions that code can encounter. This will also help ensure that code handles errors gracefully. Writing unit tests for all new code helps catch bugs early and prevent them from propagating into the codebase.

Unit tests can, and in many cases should, drive the development process. Writing unit tests for the functionality to implement helps ensure it's built correctly. The following example creates a test case that checks whether a generic service and account are deleted appropriately:

```
import XCTest
import Security
class DeleteTest: XCTestCase {
    func testDelete() {
        let service = "myService"
        let account = "myAccount"
        let query: [CFString: Any] = [
            kSecAttrService: service,
            kSecAttrAccount: account,
            kSecClass: kSecClassGenericPassword,
        ]
        let status = SecItemDelete(query as CFDictionary)
        XCTAssertTrue(status == errSecSuccess || status == errSecItemNot
        Found)
    }
}
```

Many applications begin with few, if any, tests. As apps grow, tests become increasingly important. Not only do they prevent escape defects, or bugs, but they also help in troubleshooting. The more granular the tests, the more helpful. As functions or methods are broken apart when code grows, sometimes it becomes arduous to continue to have as much test coverage as possible. However, this is one of those places where tools like OpenAI can be helpful.

Using AI to Build Tests Automatically

Unit tests are one of those things that evolve over time. When talking about test coverage, anyone that thinks they have full coverage is kidding themselves, but it's not uncommon for workflows to involve just building some stuff and then going back and filling in the unit tests before bringing in a second person. And yet developers know to do them. It's then possible to get help with this specific part of projects from sites like Upwork.

There's been a lot of talk recently about AI eating the world, so let's see how much of this kind of thing AI could get developers for free across languages. Google Bard and other platforms can be used to add the ability to automate other code maturity tasks in Swagger, Postman, and Google Cloud Functions/Lambdas. There can be syntax errors when trying to generate raw code, which tend to take longer than writing it manually, but for a lot of the other adjacent tasks, there are plenty of productivity gains from these kinds of platforms.

Further, the author of this book, while working on this section, built an Xcode extension called TestFairy, available for download at https://github.com/krypted/TestFairy. TestFairy creates an item in the Editor menu that says Generate Tests (Figure 2-4). That will create test cases for any code that's been highlighted.

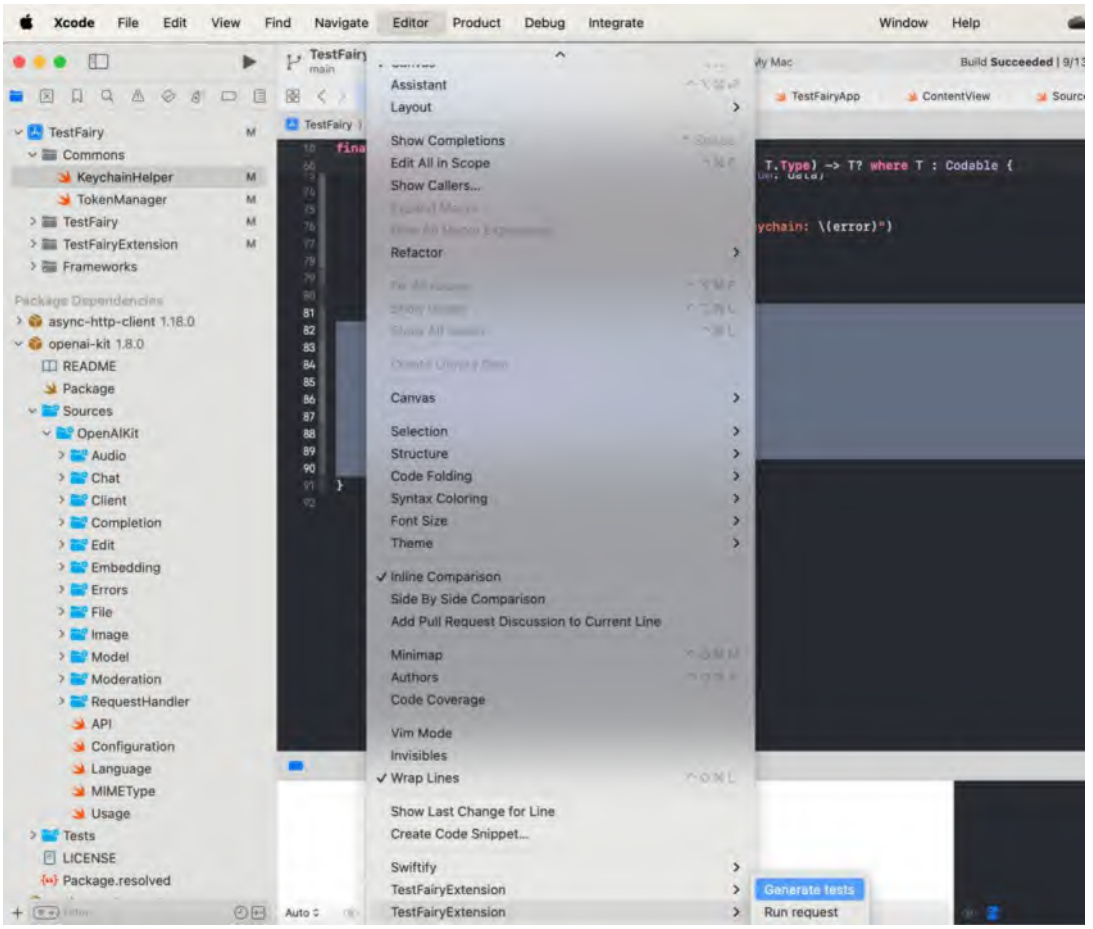

Figure 2-4. *The TestFairy Xcode extension in the Editor menu*

There are now other Xcode extensions that perform this same type of functionality, but it should illustrate that, while test cases can be time consuming, they don't have to be!

CHAPTER 3

Extensibility

Modern programming languages give developers a lot "for free." What we mean by this is software developers once upon a time had to build their own libraries to do things like work with certificates, convert data from one format to another, or perform a GET request on some web endpoint. Some will choose to still write their own code to do these common tasks, and sometimes because they need to do so in a bespoke way. However, most will use code developed by others; code that extends the use of the base libraries that come with a language.

Swift is no different and, as with many a mature language, is extensible in a number of ways. This is to say that there are pre-built packages, extensions, kits, and APIs that allow developers to get more done, faster. The hard part can be when and how a developer uses the different methodologies to use code others make available. That typically includes code from Apple, code sold by third parties, and code made available with an open source license by other developers. The way code is brought into projects, as it has since the idea became standard, is reflected with various design patterns developers follow.

When writing about design patterns, expect a lot of disagreement, especially when talking about abstractions that aren't exactly canonical, and there's plenty of disagreement even when they are. There are a lot of ways to bring pre-written code and functionality into a project in Xcode. Going back in time, there were two primary abstractions in how that was done. Those are frameworks and libraries, still used today. Within those, there are other abstractions that have evolved, which include concepts like software development kits (SDKs), applications (compiled and not compiled, or even JIT compiled), extensions, and kits. We'll go through each in more detail, but most developers will provide different responses based on their lived experiences in when and how to leverage pre-written code or distributing pre-written code. This is even true amongst engineers that write the languages themselves.

C. Edge et al., *Beyond Scripting*, https://doi.org/10.1007/979-8-8688-3035-8_3

Because this chapter spans many related concepts, the goal is not to memorize every abstraction. The goal is to understand when each mechanism is most appropriate and what tradeoffs come with each choice.

Let's look at the highly debatable overview of what each of these are, through a Swift lens (because constraining the language helps when discussing design patterns):

- *Library*: Pre-written code that can be imported into a project. Comes in two forms, dynamic libraries, which are less of a thing than they used to be, and static libraries, which in most languages are imported into an application when called. This is less of a thing with system libraries (e.g., in Python) but still a thing. In Xcode, whether something is static or dynamic is controlled by target/dependency build settings.
- *Framework*: Opinionated pre-written code that isn't just called but then imposes structure around how you write your code. The developers design the final product in a way, and the framework can contain multiple libraries. Developers can define the headers and methods available (public) and which are only accessible to the framework (private).
- *Extension*: Components, methods, types, initializers, subscripts, and protocols – executable files bundled in an app file, a keyword that allows developers to extend a type. These are discrete factored pieces of code that can be called by others.
- *Apps and app extensions*: Completed products available for distribution (in whatever form that means given the state of an app). They are done, and most wouldn't nest apps within apps or extensions in extensions, really (although extensions are nested in apps). System extensions are user-space replacements for many old kernel-extension use cases.
- *SDK*: A collection of artifacts that can be used to design a specific type of application. Usually includes libraries and/or frameworks but can also include other artifacts, like documentation.
- *Kit*: Apple's manifestation of an SDK, which goes back to the NeXT era.

- *Package*: A folder of folders and files with a specific format and a manifest. Can contain libraries of symbols. These are conceptually very clean and effectively canonized.
- *Plug-in*: Similar to an app extension but one we build into apps for others to then consume.

Use the following quick guide when terms feel overlapping:

Abstraction	Use when...
Library	You need reusable functions/types while your own code still controls the program's execution flow
Framework	You want a broader, opinionated architecture that also provides reusable APIs
Extension (Swift-type extension)	In your code, you want to add behavior to an existing type
App extension	You need to expose focused functionality to the system or host apps
SDK or kit	You are integrating a full platform/service with tooling, docs, and domain APIs
Package	You want modular, versioned, shareable dependencies managed through Swift Package Manager
Plug-in	You are extending a host app that supports third-party plug-in loading points

One important concept to consider when considering each of these is what is commonly referred to in computer science as the Inversion of Control.

Inversion of Control

Ultimately, the difference between frameworks and libraries lies in a programming concept called Inversion of Control, or IoC. IoC is a design pattern where the control of objects is transferred to a generic framework. In traditional procedural programming, custom code that expresses the purpose of code calls into reusable libraries helps a programmer take care of generic tasks, like a CIDR calculator, but in more complex environments where Inversion of Control is flipped, a framework that calls into the custom, or task-specific, code cedes object control to the developer or the original code

being imported. Think of the Apache Software Foundation, where Stefano Mazzocchi popularized the term in 1999 – they could provide frameworks about how HTML would be rendered without ceding control of all the objects in an increasingly complicated web server.

There are many different ways to implement IoC. One common way is to use a dependency injection framework. A dependency injection framework provides a way to inject dependencies into classes at runtime. This allows classes to be decoupled from their dependencies, making them more flexible and easier to test. Another way to implement IoC is to use a service locator. A service locator is a class that provides access to a set of services. This allows classes to request services from the service locator without knowing where the services are located. This makes the classes more loosely coupled and easier to test.

There are also some drawbacks to the IoC design pattern, which are commonly recurring themes in layers of obfuscation. IoC can add some complexity to code. This is because developers need to learn how to use IoC frameworks and how to inject dependencies into your classes. IoC can also increase coupling between classes. This is because classes that use IoC frameworks are more likely to depend on specific implementations of interfaces. Finally, IoC can decrease performance in some cases. This is because IoC frameworks may add some overhead to your code.

Frameworks vs. Libraries

In software development, a framework and a library are both pre-written code that can be used to speed up development and add functionality to an application. However, there are key differences between the two.

A library is a collection of reusable code that provides a specific set of functionality. Libraries are typically used to perform common tasks, such as string manipulation, data manipulation, or graphics rendering. Libraries are typically written in a generic way so that they can be used by a variety of different applications.

Frameworks are usually more complete. Frameworks provide a set of classes, interfaces, and other tools that can be used to create an entire application. Frameworks typically provide a specific way of developing an application, and they often come with a set of design patterns and best practices. Frameworks are typically written in a more specific way than libraries, so they may not be as flexible.

The choice of whether to use (or create) a framework or a library depends on the specific needs of the application. If the application needs to perform a specific task, such as string manipulation, then a library may be the best option. If the application needs to be developed quickly or needs to follow a specific set of design patterns, then a framework may be the best option.

Within libraries, there are static and dynamic libraries. Static libraries are linked into the executable file at compile time, while dynamic libraries are linked at runtime. Static libraries are a collection of object files that have been linked together. When compiling from the command line, use `-l` to request a library by name and `-L` to add library search paths. For example, to link to the math library:

```
swiftc my_file.swift -lm
```

Here, `m` refers to `libm` (the standard C math library), so `-lm` means "link against `libm`."

Once the code is compiled, the static library will be embedded in the executable file. This means that the code in the static library will be loaded into memory when the executable file is executed.

Dynamic libraries are similar to static libraries, but not linked into the executable file at compile time. Instead, they are linked at runtime. When running code, the dynamic linker will load the dynamic libraries that are needed by your code. Dynamic libraries have a number of advantages over static libraries. They allow developers to update the code in the library without having to recompile code. Second, they can be shared by multiple applications. This can save space on disk and in memory, but can be dangerous in that runtime behavior depends on the linked version at launch. In Xcode, static-vs.-dynamic behavior is configured per dependency/target (product type and embed settings).

Frameworks and Extensions

Another way to add functionality in code is an extension. A framework is usually a bundle that includes a binary, module metadata, and often resources. Framework binaries can be static or dynamic. Frameworks are typically used to provide a complete solution for a specific task, such as networking or graphics rendering. Frameworks can be imported into your code using the `import` keyword. An extension is a way to add new functionality to an existing type. Extensions are written in Swift and are compiled into your target's binary. Extensions can be added to any type, including built-in types such as String and Array.

In general, if Apple adds a complete solution for a specific task, they likely use a framework. If you need to add a small amount of functionality to an existing type, then it might be an extension. Extensions are then compiled into the same binary as the code. Frameworks provide more of a complete solution for a specific task, though. For example, the UIKit framework provides a complete solution for building user interfaces for iOS apps. Frameworks can be imported into your code using the import keyword.

When importing a framework, the framework's code is added to the code. This exposes the framework's classes, functions, and other features in the project. For example, import the UIKit framework to get the UIView class to create user interfaces.

The extension keyword is a way to add new functionality to an existing type. Extensions can be added to any type, including built-in types such as String and Array. For example, to create an extension, use the extension keyword followed by the name of the type to extend. For example, the following code creates an extension for the String type:

```
extension String {
    func backwards() -> String {
        return String(self.reversed())
    }
}
```

This extension adds a new method called backwards() to the String type. The backwards() method reverses the order of the characters in a string. Extensions are a great way to add new functionality to existing types. They can add computed properties, but they cannot add stored properties because extensions cannot modify a type's stored layout.

Abstractions can be different in different languages though, especially when the determinism of type is concerned. For example, in Python, a protocol is a set of abstract methods that a class must implement in order to be considered a member of a particular type. Protocols are similar to interfaces in other programming languages, but they are more flexible. Protocols can be used to define the behavior of a class without specifying the implementation details.

Extensions and delving into protocols and various forms of methods, while important concepts, is more about object oriented programming concepts. The term "extension" can also be used to define an application extension as well, thus making that definition more pertinent to this article. In Swift, an app extension is a separate

executable bundle that can be used to add new functionality to an existing app. App extensions can be used to add new features, improve performance, or extend the capabilities of an app. There are many app extension points; common examples include

- *Share extensions*: Share extensions add custom sharing options to the system share sheet.
- *Action extensions*: Action extensions provide custom actions on content in supported host apps.
- *Widget extensions*: Widget extensions create glanceable widgets for system surfaces such as Home Screen/Today.
- *File Provider extensions*: File Provider extensions expose files and storage providers to the system.
- *App extensions*: Typically created as new targets inside an existing Xcode app project (`File > New > Target > App Extension`).

Once created, add the code for the extension. Xcode creates type-specific files (e.g., `ShareViewController.swift` or `ActionViewController.swift`) rather than a single default `Extension.swift`. App extensions can be used to offload focused functionality from the containing app. If developers have multiple apps and want to share code between them, shared frameworks or Swift packages are usually the right abstraction. Some extensions, like system extensions, live in predefined hierarchies, such as an app's `Contents/Library/SystemExtensions` folder.

SDKs, Kits, and Packages

A software development kit (SDK) is a set of tools, libraries, documentation, and code samples that developers use to create software applications for a particular platform. SDKs typically include everything developers need to get started, including compilers, debuggers, and code samples. SDKs are used by developers to create software applications for a variety of platforms, including operating systems, programming languages, and hardware platforms. For example, there are SDKs available for developing applications for Windows, macOS, Linux, iOS, Android, and many other platforms.

SDKs help developers create more reliable and efficient applications by providing access to pretested and optimized code (theoretically at least). They probably worked great when created (giving the benefit of the doubt) but like all tech debt can become outdated or incompatible with newer versions of the platform or programming languages used.

Another abstraction we run into is a kit, which seems to be Apple's Objective-C and Swift way of distributing SDKs. "Kit" is mostly an Apple naming convention (e.g., UIKit, CloudKit, SpriteKit). In practice, these are usually frameworks and related APIs/resources oriented around a domain.

Kits typically provide all of the code and resources you need to get started, so you don't have to start from scratch, but may not be compatible with all versions of Swift, enforce certain entitlements requirements in apps, and restrict the build target (OS versions) available. Essentially "Kits" are a type of SDK.

A Swift package is a collection of Swift code, resources, and metadata that can be used to develop a specific type of application. Packages can be created by Apple, third-party developers, or private teams and often come with different licensing requirements. Packages can be used to speed up game, web, and network programming or whatever else the developer creates or exposes. Packages can also include artifacts and resources such as images, sounds, fonts, and even documentation. Unlike the "*Kit" naming convention, Swift packages are formally defined by Swift Package Manager:

- *Packages are self-contained*: A Swift package contains everything you need to use it, including the code, resources, and metadata (although let's be honest, ymmv here). This makes it easy to install and use packages in your projects.
- *Packages are versioned*: Swift packages are versioned, which means that you can specify which version of a package you want to use in your project. This helps to ensure that you are using the latest and most stable version of a package.
- *Packages are modular*: Swift packages are modular, which means that you can use parts of a package without having to use the entire package.
- *Packages are discoverable*: Many Swift packages are discoverable on tools like the Swift Package Index, while others are private/internal.

Many developers like to bundle discrete bits of functionality into packages. This helps modularize and improve unit testing. Just be careful not to have too many, or it can take way longer than necessary to compile.

Plug-Ins

One final way many developers abstract tools is a plug-in. Plug-ins can be used to add new features, improve performance, or extend the capabilities of an application. Some common uses for plug-ins include

- *Add new features to an application*: Plug-ins can be used to add new features to an application that would not be possible to add without modifying the application's code. For example, a plug-in could be used to add a new menu item to an application or to add a new toolbar.
- *Improve performance*: Plug-ins can be used to improve the performance of an application by offloading some of the application's functionality to the plug-in. For example, a plug-in could be used to handle image processing or to handle network requests.
- *Extend the capabilities of an application*: Plug-ins can be used to extend the capabilities of an application by adding new functionality that is not available in the base application. For example, a plug-in could be used to add support for a new file format or to add support for a new protocol.

When thinking about creating a paradigm that allows for plug-ins, there are a few things to keep in mind. Not all applications support plug-ins, so check the documentation for the application to make sure it is supported (we won't go into using private APIs to plug into apps here). Also decide what kind of plug-in to allow for. What functionality and what limits on functionality will be provided and how. This might mean a command-line interface, allowing developers to deep link into a URI of the app, etc. Consider Logic and how plug-ins for that can be run in another tool like Hindenberg. Good plug-in toolkits are extensible and reusable. Unit tests, while necessary, are incredibly difficult if/when you expose too much. Yet, creating a plug-in can be a great way to add new functionality to an existing application.

Plug-ins can mean different things in different languages. Legacy Apple developers will note that an Objective-C plug-in is a software component that can be added to an Objective-C application to extend its functionality. Plug-ins can be used to add new features, improve performance, or extend the capabilities of an Objective-C-based application. They are often delivered as loadable bundles (and sometimes dynamic libraries) and are commonly discovered from plug-in-specific locations such as an app's `PlugIns` directory.

Plug-ins can be used to add new features to an application in a number of ways. For example, a plug-in could be used to add a new menu item to an application or to add a new toolbar. Plug-ins can also be used to improve the performance of an application by offloading some of the application's functionality to the plug-in. For example, a plug-in could be used to handle image processing or to handle network requests. Plug-ins can also be used to extend the capabilities of an application by adding new functionality that is not available in the base application. For example, a plug-in could be used to add support for a new file format or to add support for a new protocol.

Humans, Abstractions, and Design Patterns

Ultimately, many of the above concepts can be interchangeable in a way. Where one developer will use a Swift Package, another might create an extension so they don't load too much code into memory when two different applications are used. Today, five developers gave me conflicting descriptions and use cases for when they use the above design patterns to implement existing functionality or make options available. There were few best practices but many worst practices. For example, when to use a Swift Package vs. create an extension. I would personally create an extension if I had reusable bits of code that I wanted compiled in such a way that they only ran in memory once. This might be (and likely would be) derived from a package, but some things you just wanna' run once. Some extensions are necessary, like Autofill, which is arguably an app that is being extended into WebKit. If a developer has only worked in a scenario that calls for one of these they might not ever consider the other.

Computer science and formal declarations of some of these design patterns exist, but use comes down to familiarity unless, as with Swift Package Manager, a vendor has canonically defined use. Even here, where some tend to use packages one way, others might release a simple library as a Swift package. Similar in concept to creating a GitHub repository vs. a gist, sometimes. Take the term design pattern, an abstraction of ideas.

A design pattern is a reusable solution to a commonly occurring problem in software design. Design patterns are typically described in terms of their intent, structure, and participants.

Some might think of a pattern that defines the skeleton of an algorithm, deferring some steps to subclasses as the meaning of that term, when in fact that's a Template Method, one of many formalized design patterns. Others include

- *Singleton*: A pattern that ensures that only one instance of a class exists in an application
- *Factory*: A pattern that creates objects without exposing the instantiation logic to the client
- *Adapter*: A pattern that converts the interface of one class into an interface that another class expects
- *Strategy*: A pattern that allows you to vary the algorithm used to perform a task without affecting the clients that use the algorithm

Ultimately, what matters is that rather than arbitrarily use the terms, we define what they formally mean to us – and when various pluggable structures are used when possible. Much as spoken languages are a design pattern to communicate human thought with fidelity, the lexical references we make to describe programming techniques and design patterns communicate how we write code. The keyword there is with fidelity. Little has changed about the fidelity of communication (no matter how concise we try to be) since Claude Shannon's great work *A Mathematical Theory of Communication*, available at https://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf. Generations later though, we have gotten to a point where we can describe vast bodies of knowledge about programming very concisely, like "let's write a framework to do that." Provided of course, we are all on the same page about what a framework is.

Controlling Access with Entitlements

Apple has provided developers with a number of ways to extend functionality. Over time, the developers that wrote ways for apps to talk to specific features realized that some apps might take advantage of the access they had in ways users hadn't intended. Thus, to protect privacy and the security of the frameworks and APIs exposed, Apple introduced the idea of entitlements.

Entitlements provide a way to specify the capabilities and permissions granted to your app when it runs on a device or interacts with specific system resources. They allow you to define the access and privileges your app needs to perform certain tasks or access protected resources. Entitlements are code-signing claims represented as key-value pairs in a property-list format (often a `.entitlements` file at build time, then embedded in the app signature). Each entitlement corresponds to a specific feature or capability that your app requires or wants to access.

Xcode can add entitlement keys when you enable corresponding capabilities. For example, enabling Push Notifications and using a provisioning profile that supports it will result in the `aps-environment` entitlement in the signed app. Developers will need to manually add some entitlements to an app, though, or request approval for restricted capabilities. To add a custom entitlement, create an entitlements file in Xcode and specify the necessary entitlement keys and values. Some of the more common manual entitlements include App Sandbox entitlements, keychain access groups, iCloud containers, etc.

Add, modify, and remove entitlements through the app's target settings in Xcode. To do so, select the target for your app and navigate to the "Signing & Capabilities" tab. Expand the "Capabilities" section to see a list of available entitlements. Toggle the switches next to each entitlement to enable or disable them as per your app's requirements. For manual entitlements, create an entitlements file, specify the necessary keys and values, and reference the file in your target's settings.

Entitlements are granted per app. Entitlements are closely tied to an app's identifier (App ID) and provisioning profiles. Each app ID is associated with specific entitlements, and provisioning profiles ensure that an app is signed with the appropriate entitlements for distribution. Make sure an app's entitlements match the capabilities enabled in provisioning profiles to avoid signing and entitlement conflicts.

See What Entitlements an App Has

It's also possible to check what entitlements an app has. During debugging, use code-signing tools (such as `codesign`) to inspect the signed entitlements for a built app. In code, entitlement checks are done with Security framework APIs, not `ProcessInfo`.

Application extensions are separate bundles that expose focused functionality to host apps or the system. Frameworks and packages are imported independently. For example, an app can ship a Quick Look thumbnail extension (`com.apple.quicklook.`

thumbnail) to provide previews in system contexts. Click Privacy & Security in System Settings, then Extensions, and then Quick Look to see non-Apple apps that expose that capability.

These are loaded during plug-in discovery. Third-party app extensions are typically packaged in the containing app bundle (e.g., .../Contents/PlugIns/*.appex on macOS), while Apple-managed extension implementations can also exist in system locations such as /System/Library/ExtensionKit/Extensions. There are tons of extensions. Some integrate with UI surfaces and others allow developers to work with files more easily (e.g., file provider extensions used by cloud-storage apps). In the Extensions screen of System Settings, click Finder extensions to see those that are loaded (including if it's installed, OneDrive). Sharing sheets, Notifications, Photo Editing, and others can be seen in Extensions as well. Most (although not all) will require users to explicitly grant access to the necessary resources to load.

System Extensions are a replacement for kernel extensions and are typically used to change the way the system responds to various events. Some have higher barriers of entry, such as individual meetings to make sure the power they give the developer isn't used for unintended purposes. Another barrier of entry is that users can disable them and/or have to have them enabled via an MDM as a part of a command or profile. To see a list of active System Extensions, use the systemextensionsctl command with a list verb to see the running extensions:

```
systemextensionsctl list
```

Some extensions also fall somewhere between what a System Extension might be used for and what app extensions are meant for. Some can be enabled programmatically, such as with:

```
pluginkit -e use -i io.MyAwesomeApp.AppExtension
```

Many cannot. One of these might be the Credential Provider extension. As these are managed by PluginKit, the pluginkit command can be used to see them loaded and provide metadata about them. For example, rather than use the Privacy & Security System Settings ➤ Extensions to see the Password AutoFill, we can check the status with pluginkit (which resembles Jorge but for extensions instead of hardware). The command to see this information would be pluginkit with some switches – for example, the following shows all extensions:

```
pluginkit -mAvvv
```

A quick find in Terminal shows the CredentialProvider along with where the .appex bundle is located:

```
io.myawesomeapp.myapp-macOS.CredentialProvider(0.1.0)
Path = /Applications/myapp-macOS.app/Contents/PlugIns/
CredentialProvider.appex
UUID = 6688AAEE-0EC7-42B0-A55F-582AAF6F19AC
Timestamp = 2022-11-10 14:36:00 +0000
SDK = com.apple.authentication-services-credential-provider-ui
Parent Bundle = /Applications/myapp-macOS.app
Display Name = MyApp
Short Name = MyApp
Parent Name = MyApp-macOS
Platform = macOS
```

The CredentialProvider is managed by the CredentialProviderExtensionHelper and the AppleCredentialManagerDaemon. The PluginKit binary is pkd (invoked by launchd and if all is well consuming less than maybe a dozen megs of memory). Inside the CredentialProvider.appex bundle there is an info.plist with metadata about the credential provider and any app that loads an extension should have a corresponding /Applications/MyApp-macOS.app/Contents/_CodeSignature/CodeResources file that lists the extensions loaded:

```
cat /Applications/MyApp-macOS.app/Contents/_CodeSignature/CodeResources
```

Here, there are keys for resources and plug-ins loaded. That credential provider is then loaded in user space at `~/Library/Application Support/com.apple.AuthenticationServices/CredentialProviders` (access can still be constrained by process context, sandboxing, and other security controls). A network provider (e.g., what's used by most ZTNA solutions) might show `com.apple.networkextension.packet-tunnel`, and so looking for whether that's loaded can net a way to programmatically derive the state of those by bundle ID. Unlike kernel extensions, these are imported and exposed as well-documented APIs for developers. To interrogate extensions (rather than walk through info plists and the like):

```
pluginkit -m -v -i io.MyAwesomeApp.AppExtension
```

Note Most of this also applies to iOS simulators but you can't really get to it on a running device.

Some of these require entitlements. To see a list of entitlements for many of these, see Jonathan Levin's entitlement database at `http://newosxbook.com/ent.jl?osVer=MacOS15`. It doesn't include which third parties have access to entitlements but does lay out how Apple uses many internally. Entitlement enforcement comes from code-signing and security policy checks and complements System Integrity Protection.

System Integrity Protection

System Integrity Protection (SIP), also known as "rootless," is a security feature introduced in OS X El Capitan (10.11) and later versions of macOS. SIP is designed to protect critical system files and directories from unauthorized modifications, even by users with administrative privileges or processes running as the root user. SIP provides the following:

- *Core functionality protection*: SIP safeguards critical system directories and files, including `/System`, `/bin`, `/sbin`, and much of `/usr` (with notable exceptions such as `/usr/local`). It prevents unauthorized modifications to these protected system components.
- *Restricted root user access*: SIP restricts the capabilities of the root user, which is the superuser account with unrestricted access to the entire system. With SIP enabled, even the root user cannot modify or tamper with protected system files and directories.
- *Limited kernel extensions*: Third-party kernel extensions are deprecated and heavily restricted. Where still supported, they require Apple-compliant signing/notarization and additional approval/security requirements.
- *System file and directory lockdown*: SIP prevents both write and deletion operations on protected system files and directories, including critical system apps, frameworks, and libraries. This protection ensures the integrity and stability of the operating system.

- *Enhanced system security*: By enforcing system integrity, SIP helps defend against malware, unauthorized modifications, and tampering attempts that could compromise the stability and security of macOS.

While rarely necessary in production, some developers may find that it's easier to troubleshoot issues that arise from entitlement issues in apps by disabling SIP temporarily. By default, SIP is enabled on macOS systems, and it is recommended to keep it enabled for optimal security. To be clear, disabling SIP should only be done when absolutely necessary for specific use cases, such as debugging low-level system components or installing certain software that requires modification of protected system files. To enable or disable SIP, boot a Mac into Recovery mode, open the Terminal, and use the csrutil command:

```
csrutil disable
```

Swap the disable with enable to turn SIP back on. Another option that's helpful when debugging system extensions can be developer mode. Depending on macOS version and hardware, enabling this may require additional security configuration:

```
systemextensionsctl developer on
```

If using an Apple Silicon (arm64) machine and debugging driver extensions, it might also be necessary to enable NVRAM options as follows:

```
sudo nvram boot-args=-arm64e_preview_abi
```

Once booted and able to interface with system extensions directly, consider a tool like LLDB. To enter the LLDB interactive mode, simply run lldb:

```
sudo lldb
```

Now from the interactive mode, attach to a process to see what's happening when it's run:

```
process attach --pid 8766
```

The above attaches to a specific PID, but it's also possible to use --waitfor with it when attaching to the next instance. To attach to an app, just run lldb followed by the .app bundle URI. For example, if putting test apps into /Compiled, to attach to an app called myextension.app:

```
lldb /Compiled/myextension.app
```

It's also possible to then see any breakpoints as well (from that LLDB interactive mode):

```
breakpoint list
```

So let's say an extension is invoked in line 87 of file blocker.c, create a breakpoint there:

```
breakpoint set --file blocker.c --line 87
```

This sets the breakpoint for each method that implements that line, or its class, as well. Once the breakpoint is hit, use the thread verb followed by continue to proceed:

```
thread continue
```

This brings up threads. From within LLDB, use the thread verb followed by list to see the state the process is in:

```
thread list
```

Or use thread with backtrace to see those:

```
thread backtrace
```

Also check out watchpoint and frame in LLDB. LLDB stands for low-level debugger. Developers often use it to debug their own code, but it's also useful with other constructs, like libraries.

Libraries

A shared library is a file that contains code that can be shared by multiple programs. Shared libraries are often used to provide common functionality, such as mathematical functions or file I/O. To use a shared library in Swift, link it to a program. This is done in Xcode by going to the Build Settings tab for your project and adding the shared library to the Linked Frameworks and Libraries section.

Once linked, the library can be imported into code using the import keyword. For C math functions on Apple platforms, that typically comes through the Darwin module:

```
import Darwin
```

Once a shared library is imported, the functionality can be used in code. For example, the following code uses the sin() function to calculate the sine of a number:

```
let angle = Double.pi / 6.0
let sine = sin(angle)
print(sine) // 0.5
```

When using shared libraries make sure to be aware of the different versions of the library that are available. Make sure that tools are compatible with the version of the library in use. As mentioned, shared libraries are just one way to share code between apps. Another way to tap into the work done by others is to use a Swift package.

Swift Packages

A Swift package is a collection of Swift source files, resources, and metadata that defines a cohesive unit of functionality. Packages can be used to share code between projects or to create libraries that can be used by other developers. They're also a way to break up large projects into smaller components, even if those components won't be shared. This allows for

- *Reusable code*: Swift packages make it easy to share code between projects. This can save you time and effort when you are developing your apps. For example, if you have written a library of functions that you use in multiple projects, you can create a Swift package for that library and then install it in each of your projects. This will save you the time and effort of having to copy and paste the code into each project.
- *Libraries*: Swift packages can be used to create libraries that can be used by other developers. This can help you to share your code with the wider community and make it easier for other developers to build great apps. For example, if you have written a library of functions that you think other developers might find useful, you can create a Swift package for that library and then publish it in a Git repository or package registry. This will make it easy for other developers to find and use your library.

- *Version control*: Swift packages use version control to track changes to your code. This makes it easy to track changes to your code and to revert to previous versions if necessary. For example, if you make a change to your code that you later decide was a mistake, you can easily revert to the previous version of the code.
- *Dependency management*: Swift packages make it easy to manage dependencies between projects. This means that you can be sure that the versions of the libraries that you are using in your projects are compatible with each other. For example, if you are using a library that is updated frequently, you can be sure that the latest version of the library will be compatible with the version of the library that you are using in your project.

To create a Swift package

- *Open terminal*: Launch the Terminal application.
- *Choose a directory*: Navigate to the desired directory using the `cd` command in Terminal.
- *Create the package*: In Terminal, run the following command to create a new Swift package:

```
swift package init --type executable
```

This command initializes a new Swift package with the executable template. To create a library package, replace `--type executable` with `--type library`.

After running the command to create a package, a new directory will be created with the same name as the package. Inside that directory will be the basic structure of the package, including the `Sources` directory containing the main Swift source file. Open the Package.swift file located in the root of your package directory. This is where modifications to various details such as the package name, supported platforms, dependencies, and targets will be made.

Within the `Sources` directory, there will be a Swift file with the same name as the package. Start writing code in this file or add additional source files as needed. Once done, to build the Swift package, go to the Terminal, navigate to your package directory, and run the following command:

```
swift build
```

This command compiles the package and generates the executable or library output. To then test the package, use the following command:

```
swift test
```

This command executes the test cases defined within your package, if any have been defined.

The Swift Package Manager provides additional commands and functionalities to manage dependencies, inspect manifests, and automate builds/tests. In modern workflows, integrate with Xcode by opening `Package.swift` directly or adding package dependencies in Xcode (the older `generate-xcodeproj` flow has been removed).

Call a Swift Package

To use a Swift package in a project, add it as a dependency. To do so, open Xcode, click the File menu and then select Add Packages. Search for the package to add. Once added into an Xcode project, add the package as a dependency. This is done by importing it into code using the import keyword. For example, the following code imports the package RxSwift:

```
import RxSwift
```

Once the package is imported, use its functionality in code. For example, the following code creates an Observable sequence that emits the numbers from 1 to 10:

```
let observable = Observable.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

If needed, then subscribe to the Observable sequence and handle the emitted events:

```
observable.subscribe(onNext: { value in
  print(value)
})
```

This code will print the numbers from 1 to 10 to the console.

Share a Swift Package

One of the best aspects of Swift packages is how easy it is to share and consume packages made by others. There are a few common ways people share packages: local package sharing, source control repositories (e.g., git), and package registries.

To share packages the local way, share the package directory that contains `Package.swift`. In Xcode, the other developer can choose Add Packages and then add the package from a local path. Once done, add the package product to the target that needs it.

The second way is through remote package sharing, commonly achieved using a tool like git. To publish to a git repository, first push the package to a remote repository, like GitHub. Then add it as a dependency in other projects by using the File and then Add Packages option in Xcode. When prompted, enter the repository URL and select the desired version. Xcode then downloads and integrates the package.

When using packages, keep in mind that it's important to use semantic versioning (e.g., 1.2.3) to manage package updates effectively. Also keep dependencies in mind. Specify dependencies on other packages in the Package.swift manifest. Consider access levels as well. Control the visibility of package code using access control keywords like public, internal, and private. Finally, for distributing pre-compiled binaries, explore Swift Package Manager's support for XCFrameworks, which enables binary distribution.

Extensions

An app extension is a separate executable bundle that extends specific functionality of an app and makes it available to users in other parts of the system. It's a way to integrate an app's features into various system contexts and other apps, enhancing user experience and creating a more cohesive ecosystem.

Focused functionality: Each app extension should be designed to perform a specific task or set of related tasks efficiently. Extensions should have restricted access to system resources and user data compared to full apps, ensuring privacy and security. They are also triggered by the system (or at least a system event) or a host app. This is to say that they are launched by the system or a host app when needed, often based on user actions or system events.

Keep in mind that extensions are created as separate targets within your Xcode project, with their own code and resources. Consider the following common types of app extensions:

- *Widgets*: Display glanceable information or quick actions on the Home Screen or Today View.
- *Share extensions*: Add custom sharing options to the system-wide sharing sheet.
- *Action extensions*: Contribute custom actions to other apps' action sheets or menus.
- *Notification content extensions*: Customize the content of push notifications.
- *Photo editing extensions*: Add filters and effects to images within the Photos app.
- *Keyboard extensions*: Provide custom keyboard input methods for text fields.
- *Custom intents*: Allow your app to be invoked by Siri or other apps through a defined set of actions.
- *File providers*: Enable other apps to access and manage files stored within your app's sandbox.

Extensions make an app's features more accessible and integrated throughout the system. When used appropriately, they can provide a convenient way for users to interact with your app in different contexts. This extends the reach, visibility, and potential use cases for apps. To create an extension, from Xcode, create a new app extension target within a project and then implement the required extension point protocols to define its functionality. Just keep in mind it's necessary to handle communication with the host app or system using a limited set of APIs. Or if consuming one of Apple's extensions, make sure the extension adheres to Apple's guidelines for performance and user experience.

Entitlements

As mentioned, entitlements are special permissions that can be granted to apps. They allow apps to access features and resources that would not normally be available to them. For example, an app might need entitlements for App Sandbox configuration, App Groups, iCloud containers, or Network Extension capabilities. To use entitlements, first add them to an app's capabilities. This is done in Xcode from the Capabilities tab in project settings, as seen in Figure 3-1.

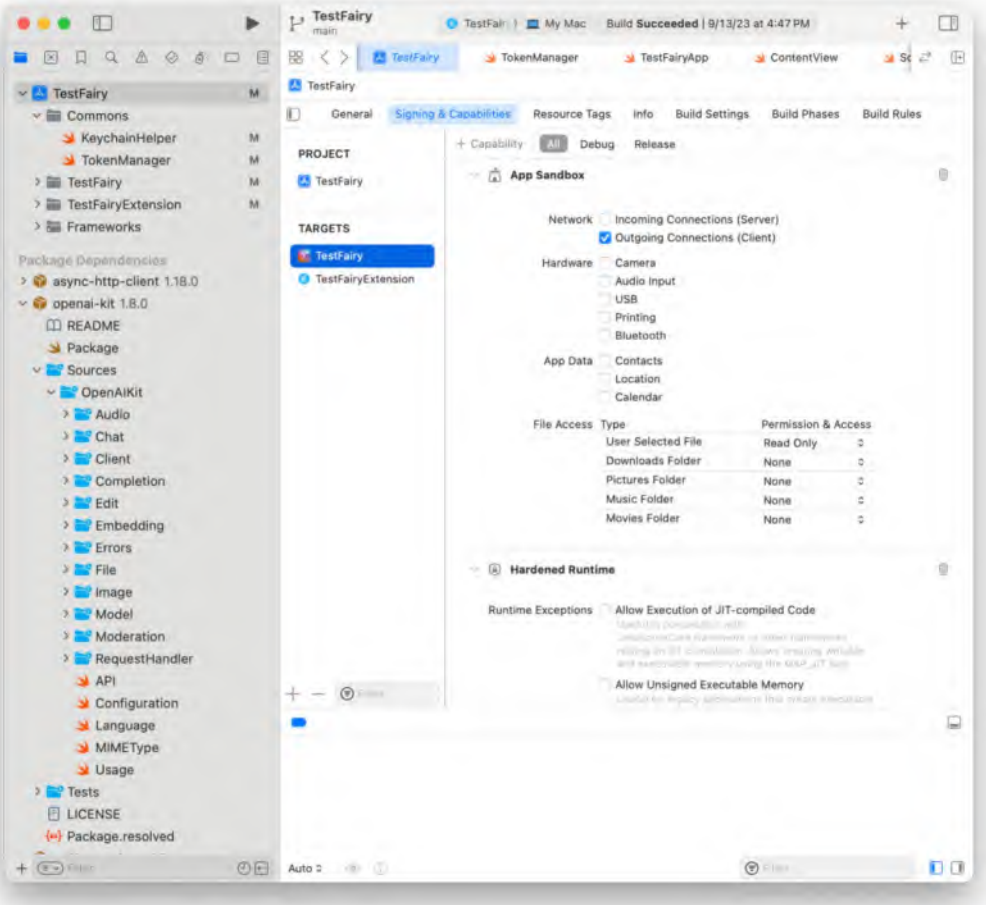

Figure 3-1. *Entitlements*

To inspect signed entitlements at runtime, use Security framework APIs such as `SecTaskCopyValueForEntitlement`. For example:

```
import Security

if let task = SecTaskCreateFromSelf(nil),
   let sandboxValue = SecTaskCopyValueForEntitlement(task, "com.apple.
   security.app-sandbox" as CFString, nil) {
    print("Sandbox entitlement:", sandboxValue)
}
```

Location access, camera access, and similar privacy controls are permissions, not entitlements. Check those using the relevant framework APIs. For location:

```
import CoreLocation
func checkLocationPermission() {
    let manager = CLLocationManager()
    let status = manager.authorizationStatus
    if status == .authorizedAlways || status == .authorizedWhenInUse {
        // The user has granted location permission.
    } else {
        // Location permission has not been granted.
    }
}
```

Use signed-entitlement checks for entitlement keys, and use each platform API (e.g., Core Location or AVFoundation) for user permissions.

Using Apple Extensions

To use an Apple app extension in a Swift project, first create a new extension target as described earlier, but in the "New Target" dialog box, select the "App Extension" option and choose the type of extension to create. Once the extension target is created, add the necessary code to the extension. The code for the extension will depend on the type of extension being created. For example, if creating a keyboard extension, add code to handle keyboard events.

Each extension point comes with its own protocol that defines the functionality and how to implement it. Getting familiar with the protocol methods involves reading

the documentation. Once that's read, start building the logic in Swift. For example, a share extension would involve handling the itemsToShare parameter and performing operations based on the shared content.

Depending on your extension point, it likely needs to communicate with the host app to exchange data or trigger specific actions. Xcode provides APIs like `NSExtensionContext` and shared containers (e.g., App Groups) for this purpose. Ensure the communication flow is efficient and secure.

Let's look at the full lifecycle of creating an app extension and, in this case, a share extension to do something that has long been a common task for MacAdmins: edit a property list. First, set up the Extension Target by clicking File, then New, and then Target from within an app in Xcode. Since this will be a share extension, select Share Extension and click Next. Then provide a name for the extension (e.g., PlistEditorExtension) and choose `NSExtensionPrincipalClass` as the principal class (if applicable to the template you choose). Then click Finish.

Next, let's implement it in SwiftUI or UIKit. In this example, let's use a view controller in UIKit. To do so, open the `ShareViewController.swift` file (or your designated view controller file) and import the necessary frameworks by using the import keyword followed by their name in the beginning of the file:

```
import UIKit
import UniformTypeIdentifiers
```

Next, add a property to store the property list data:

```
var plistData: [String: Any]?
```

Next, implement the `SLComposeServiceViewController` hook `didSelectPost()` to handle user interaction:

```
override func didSelectPost() {
    guard let item = extensionContext?.inputItems.first as?
    NSExtensionItem,
          let itemProvider = item.attachments?.first(where: {
              $0.hasItemConformingToTypeIdentifier(UTType.propertyList.
              identifier)
          }) else {
        return
    }
```

```
    itemProvider.loadItem(forTypeIdentifier: UTType.propertyList.
    identifier, options: nil) { [weak self] item, error in
        guard let self = self else { return }
        if let data = item as? Data {
            do {
                self.plistData = try PropertyListSerialization.propertyList
                (from: data, format: nil) as? [String: Any]
                // Present UI to edit the plist data
            } catch {
                // Handle parsing errors
            }
        } else if let dict = item as? [String: Any] {
            self.plistData = dict
        }
    }
}
```

Once the functionality is defined, it's time to design the editing UI. This might include text fields for key-value pairs, table views for nested structures, or buttons to save or cancel changes. Then create a button to save changes to the property list. The following stanza of code implements a method to serialize the edited data back into a plist format:

```
func savePlistChanges() {
    guard let plistData = plistData,
          let data = try? PropertyListSerialization.data(fromPropertyList:
          plistData, format: .xml, options: 0) else {
        return
    }
    // Write the data back to the original file or a new location
}
```

Then, once the save is committed, write a function to handle completion and call co mpleteRequest(returningItems:completionHandler:) when the editing is finished:

```
extensionContext?.completeRequest(returningItems: [],
completionHandler: nil)
```

Keep in mind to test the extension with different scenarios, and adhere to Apple's guidelines for action extensions.

Kits and the Apple Developer Portal

Entitlements are required to access certain APIs. Apps are published using the App Store Connect website, so it seems like that's where various entitlements would be requested. The Apple Developer Portal is where tools to build software are downloaded. That's actually where access is requested instead. That canary in the coal mine should indicate that this is going to be one of the least intuitive aspects of developing apps.

As was covered earlier for APNs (which after all uses an entitlement as well), scroll down to the footer of developer.apple.com and click Certificates, Identifiers & Profiles to get started. From the Certificates, Identifiers & Profiles page, click Identifiers. Then click the plus sign icon to see the following options:

- *App IDs*: Provides access to most of the services and it's via a provisioning profile
- *Services IDs*: Connects an app to sign in with Apple, register services identifiers (Services ID), and configure private keys
- *Pass Type IDs*: Generates Apple-issued certificates used to sign and update passes in the Apple Wallet
- *Order Type IDs*: Signs and distributes order bundles with Wallet and Apple Pay
- *Website Push IDs*: Registers push notification providers
- *iCloud Containers*: Connects apps to iCloud Storage APIs
- *App Groups*: Creates group containers to allow interprocess communication between apps (including extensions)
- *Merchant IDs*: Provides access to Apple Pay processing for websites that support Apple Pay
- *Media IDs*: Connects to the Apple Music API or ShazamKit
- *Maps IDs*: Allows embedding MapKit and information from the Maps API into a website

While it's easy to imagine admins flexing their new Swift muscles by building apps that do cool stuff, most of the work done for admins will be with App IDs (or App Groups if using keychains or sharing entitlements between apps and extensions). App IDs provide access to all kinds of APIs, though. To access them, click App IDs and then click the Continue button, as seen in Figure 3-2.

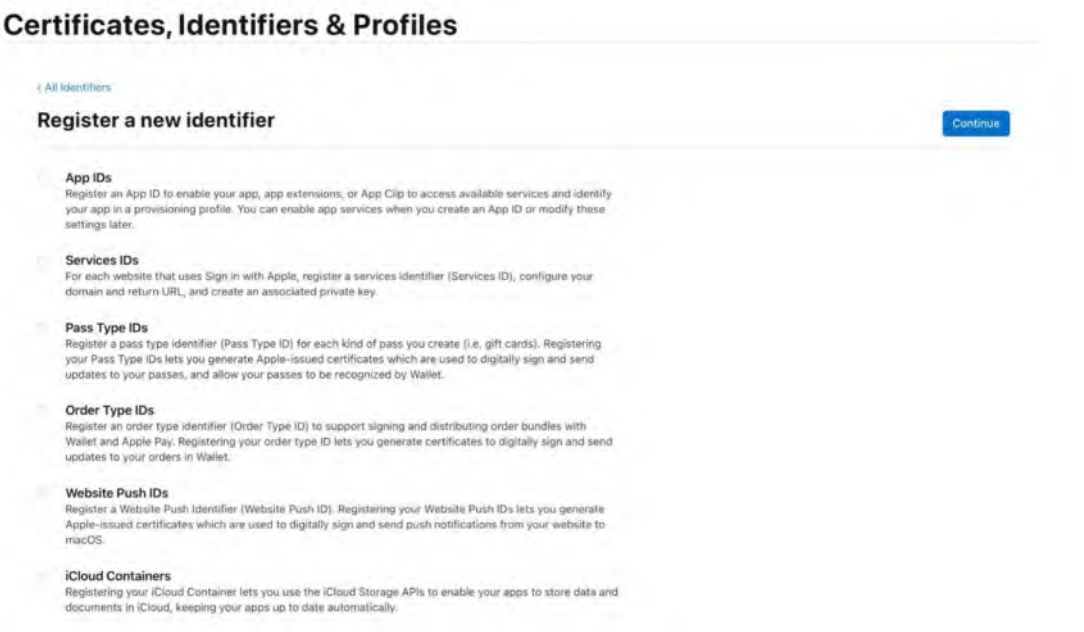

Figure 3-2. *Registering a new identifier*

The number of App IDs (68 at the time of this writing) on the next screen should provide a glimpse into the amount of work Apple has done when it comes to the intentionality about custom access to aspects of the system that can be problematic for the Apple privacy paradigm. These also don't expose certain APIs that developers will just kinda' need to know someone in Apple's Developer Relations organization to get access to. For this example, let's say it's time to build an app that analyzes network traffic that comes and goes from a machine. That would be a network extension. Start by filling in the Description (or name of an app) and the bundle identifier being used in Xcode Figure 3-3. Keep in mind that the Team ID used will need to be entered into the app, so grab that into the clipboard real quick if it hasn't already been.

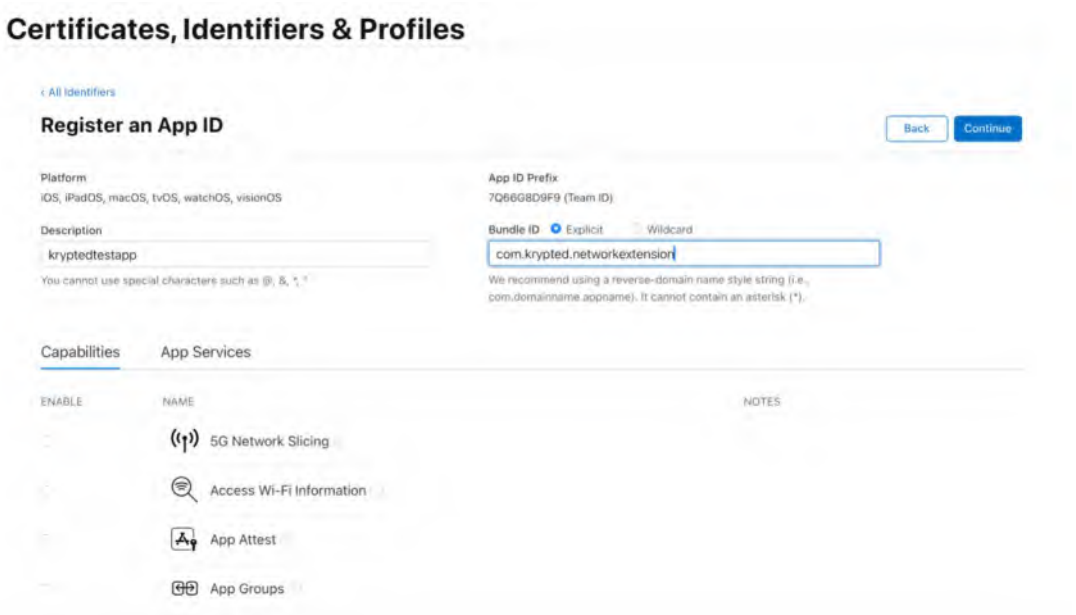

Figure 3-3. *Registering an App ID*

Scroll down to Network Extensions and check the box for it. Notice while scrolling that there are plenty of options that require special agreements or have Configure buttons beside them (Figure 3-4). Once an appropriate description, bundle ID, and any tasks for agreements or configuration are complete, click the Continue button.

Figure 3-4. *Adding capabilities to an app*

Apple is opinionated about how some of these are used and those that have a larger privacy impact (iCloud, Network, and System extensions being amongst the most potent) will likely have more opinions and valid opinions. Keep in mind, these are certificate-driven accesses, and when a developer's opinions diverge from those of Apple, they may find themselves with a revoked certificate. It would require thousands of pages to go through all of these and how to bring them in. They're wildly inconsistent and produce a barrier that may or may not be somewhat intentional. For example, if a bad developer can't figure out how to get access to CloudKit, then it's likely they can't accidentally then sync all iCloud data off to another service and steal all the things.

When there are a billion and a half iPhones out there, they are a much larger enterprise than any readers will be managing. So they require an exponentially greater amount of care. After all, when that bad developer leaks iCloud data, the headlines will just say it happened on Apple devices and probably leave out that it used a fake coupon app that users had to install to get there.

Once the service is added to the account, it's time to substantiate objects using it in code. First, make sure the appropriate Apple ID is logged in (the one the service was enabled for in the developer account) by going to the Signing & Capabilities tab for the app in Xcode, as seen in Figure 3-5.

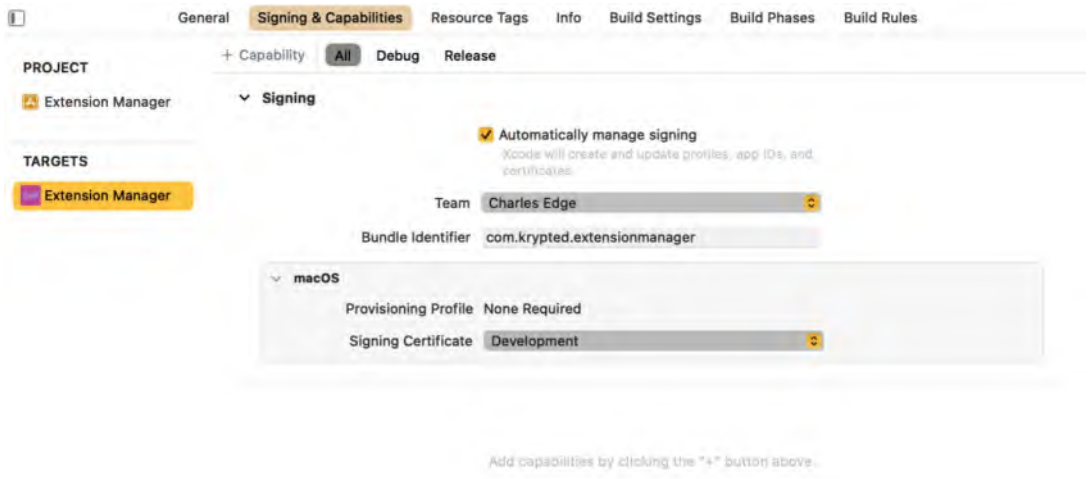

Figure 3-5. *Signing & Capabilities in Xcode*

Then click on the plus sign for Capability on the same screen and search for the service being used (in this case, "Sign in with Apple") as in Figure 3-6.

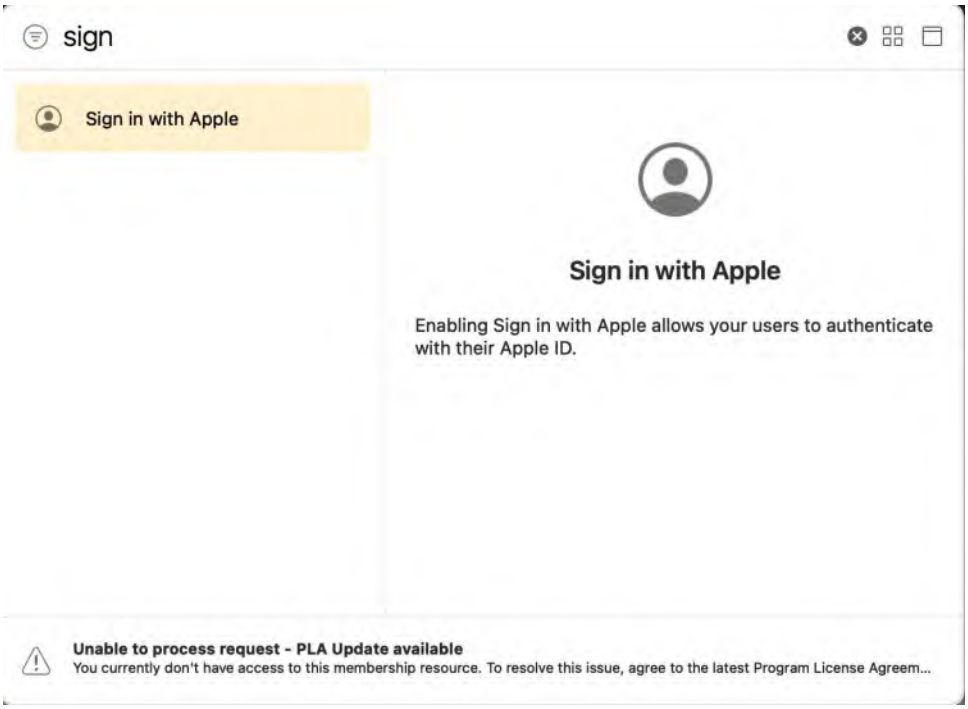

Figure 3-6. *Configure your Sign in with Apple for your app*

To continue on with the example of the Sign In With Apple service, it's usually best to then search for documentation and read all of the classes, properties, and objects that get exposed as APIs. This surfaces a button documented at https://developer.apple.com/documentation/authenticationservices/asauthorizationappleidbutton so the AuthenticationServices framework can be used to provide ASAuthorizationAppleIDButton to start the Sign In with Apple flow. Developers get a lot for free with some frameworks, like Sign In With Apple. To then create an instance of ASAuthorizationAppleIDButton, the following example checks for a finger in the bounds of the control:

```
let authorizationButton = ASAuthorizationAppleIDButton()
authorizationButton.addTarget(self, action: #selector(handleLogInWithAppleI
DButtonPress), for: .touchUpInside)
loginProviderStackView.addArrangedSubview(authorizationButton)
```

The ASAuthorizationController class is then made available, which manages authorization requests. For more on how to use that class, see https://developer.apple.com/documentation/authenticationservices/asauthorizationcontroller. From there, developers need to build logic to check if users are logged in, present buttons if not, parse the output (name, email address, etc).

All of this is fairly straightforward provided a developer knows where to find the right documentation. Sometimes the name of a class or instance property can then be searched for and code samples found to help guide the rest of the process. This is a fairly standard workflow for connecting a service from a developer ID to an app and then some code that can be used with an Apple API and how to use it. Apple APIs aren't the only ones that developers will access. A lot can be gotten for free from public APIs, like those for websites.

Interfacing with Public APIs

An API, or application programming interface, is a set of rules that allow two or more software programs to communicate with each other. It is a way for different applications to share data and functionality. APIs are used in a wide variety of applications, including

- *Web development*: APIs are used to power many of the features that we take for granted on the web, such as search engines, social media, and e-commerce sites.

- *Mobile development*: APIs are used to allow mobile apps to access data and functionality from other sources, such as weather data or stock quotes.
- *Enterprise applications*: APIs are used to integrate different enterprise applications, such as customer relationship management (CRM) systems and accounting systems.

For example, a weather app might use an API to get the current weather conditions from a weather service. A social media app might use an API to allow users to share content on other social media platforms. An e-commerce site might use an API to allow users to check their order status. Many of the companies that make apps and web apps then expose plenty of their data in the form of public APIs. Some will even provide an SDK or Swift package to import into an app to make it easier to build integrations with their services (the beauty of capitalism, right?).

REST

A REST request is a structured message sent from a client (like your web browser) to a server (like a web front-end for a database) to retrieve, create, update, or delete data. This data is often referred to as a resource and could be anything from product information on an e-commerce site to weather data from a weather API. The important parts of a REST request include

- *HTTP method*: This verb-like command tells the server what action to take with the resource. For example, GET retrieves data, POST creates new data, PUT updates existing data, and DELETE removes data.
- *URL*: This address identifies the specific resource you're targeting. Imagine it as the table number at the restaurant.
- *Headers*: These are optional metadata providing additional information about the request, like the data format (JSON, XML) or authentication credentials. Think of them as notes you might scribble on your menu order, like "allergic to nuts" or "extra gravy, please."
- *Body (optional)*: For requests like POST and PUT, the body may contain the actual data you're sending to the server. This is like handing the waiter a detailed recipe for your perfect custom dish.

REST requests come in different flavors, each with its own purpose. The ones most commonly used include the following:

- GET: The most common request, used to retrieve existing data. Imagine asking the waiter, "What's the soup of the day?"
- POST: Creates new data. Think of ordering a dish that's not on the menu and requesting the chef to whip it up for you.
- PUT: Updates existing data. It's like sending back your steak to be cooked a little more.
- DELETE: Removes data. This is like politely asking the waiter to take away an empty plate.

Armed with these, plenty of operations can be completed. For example, use GET to pull down a list of devices in an MDM, use a POST to send a wipe command to one that matches a name, a PUT to create a device, and a DELETE to remove a device once it's been retired. One of the easiest ways to test APIs is using cURL, the traditional way to perform these operations from the command line.

Use cURL to Interface with REST APIs

One challenge with communicating with REST APIs is key distribution. Another is deploying changes. These are good reasons to move some of the logic to a serverless function. For example, if there's a key on a device that allows it to perform the previously mentioned GET, POST, PUT, and DELETE operations on an MDM, a user might get that key and do things that aren't desired for a user to be able to do. However, if there's an Amazon Lambda function or a Google Cloud function, then the logic that dictates who can do what can be processed on the server side, more safely.

Security warning: Never hardcode bearer tokens, API keys, or service account credentials in Swift source files or shell scripts that are committed to source control. Load secrets from secure stores (e.g., environment variables, CI secret managers, or keychain-backed storage), and rotate them if they are ever exposed.

Google Cloud Functions offer a potent serverless platform for running code snippets on demand. cURL can be used to send HTTP requests and interact with remote servers. Before getting started with cURL, gather the following:

- *Cloud Functions account*: Sign up for a Google Cloud account and create a Cloud Function.
- *Function URL*: Locate the Cloud Function's trigger URL in the Cloud Console. It usually starts with https://.
- *Authentication*: For secure access, get an access token. These are commonly obtained via `gcloud auth print-access-token` (or `gcloud auth print-identity-token` for identity-token flows) or service account credentials downloaded as a JSON file.

Once those are available, a simple curl command to invoke the Cloud Function can be as simple as the following:

```
curl -X POST https://your-function-url -H "Authorization: Bearer YOUR_
ACCESS_TOKEN" -d '{"name": "Charles"}'
```

Breaking down the components of the above command:
-X POST: Specify the HTTP method, POST in this case, for triggering the function.

- https://your-function-url: Replace this with the actual Cloud Function URL (or better put a $ in front to make it a variable in a script).
- -H "Authorization: Bearer YOUR_ACCESS_TOKEN": Add an authorization header with the access token to authenticate or again use a variable.
- -d '{"name": "Charles"}': Send data (optional) to the function as JSON in the request body. Replace the example JSON with the specific data format, likely in JSON.

Beyond that basic incantation, it's also possible to customize the curl commands for different scenarios:

- GET *requests*: Use -X GET instead of -X POST to retrieve data from a function.
- *Query parameters*: Append parameters to your URL after the ? symbol, e.g., https://your-function-url?id=123.
- *Custom headers*: Add additional headers with -H flag, like specifying content type (Content-Type: application/json).

- *Response handling*: Capture the function's response with -o output.txt to save it to a file or -i to see detailed headers.
- *Passing files*: Use tools like base64 to encode files and send them within the request body.
- *Debugging*: Check logs in Cloud Monitoring or use -v flag with curl for verbose output.
- *Scripting automation*: Build scripts with curl commands to automate repetitive function interactions.

There's a lot that can be done with REST endpoints, but some care should be used. Avoid putting confidential data in URLs (including GET query strings), because URLs are frequently logged and cached. POST is not inherently more secure than GET; use HTTPS for transport security and keep secrets in protected headers or bodies as required. Also, make sure to keep access tokens and credentials secure.

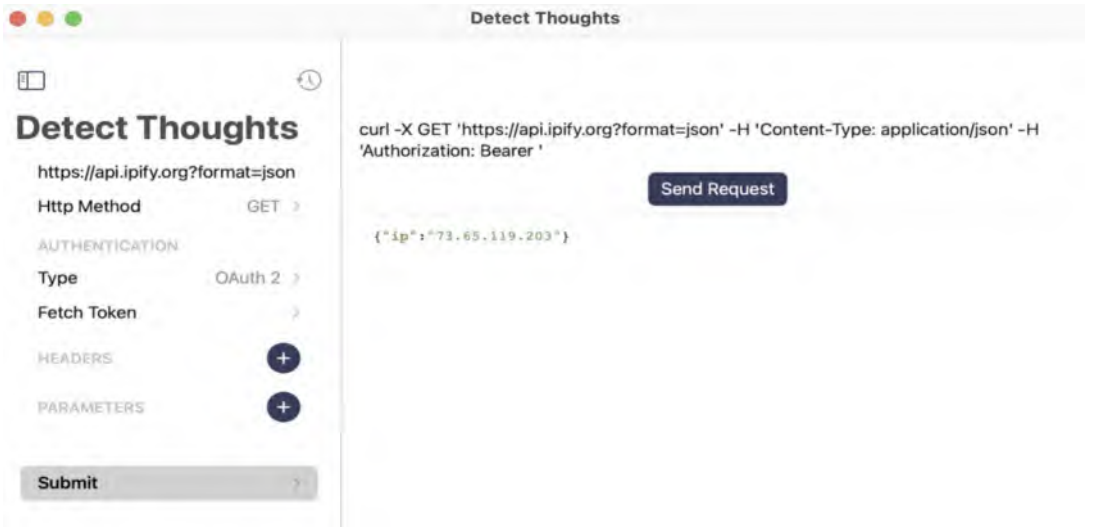

Figure 3-7. *The Detect Thoughts app*

Curl, combined with custom serverless functions is great, but the results can be different from a Mac to iOS. The authors of this book also built an app called Detect Thoughts (Figure 3-7) available at `https://apps.apple.com/us/app/detect-thoughts/id1635176129`. This can be used to get the syntax for a curl command, but also to run commands from an iOS or iPadOS device as well (or presumably VisionOS when that is available).

Note: One of the authors of this book also provided a cURL sample project at https://krypted.com/uncategorized/curl-app-framework-for-ios/ that can be used to get started with integrating web APIs into projects.

Authenticating with cURL

As mentioned, authentication to an API can be done in a number of ways. A standard is using a bearer token. In the following, a variable called BearerToken uses a simple curl with the contents of a bearer token. This is done by including data in the request for "userid" although sometimes sites might instead call this as "user" or "username" and then a password. This hits an endpoint called authenticationendpoint, which according to the site might call "auth" or "authenticate" – in this specific case, the bearer token is "id", and it's nested in there with a name of "token":

```
BearerToken=$(
  curl -s -X POST \
    -H "Accept: application/json" \
    -H "Content-Type: application/json" \
    --data '{"userid":"{userid}","password":"{password}"}' \
    https://krypted.com/api/authenticationendpoint |
  python3 -c 'import sys,json; print(json.load(sys.stdin)["token"]["id"])'
)
```

Once there's a token, it can be passed into another API via the Authorization header when connecting. The following example passes the BearerToken just captured, to an endpoint called EndpointName to https://krypted.com/api/EndpointName:

```
curl -H "Accept: application/json" -H "Authorization: Bearer
${BearerToken}" https://krypted.com/api/EndpointName
```

This is a book about Swift, but before writing Swift code, model the connections with curl or, for more advanced options, especially for public APIs, Postman.

Use Postman to Test APIs

There are plenty of ways to learn to navigate public web APIs. One of the best is a tool called Postman. Postman was designed to streamline API testing and exploration. To get Postman, visit https://www.postman.com/downloads/ and create an account. Once downloaded, install it and notice that there are collections and workspaces.

Collections are one of the best parts of Postman. A Postman collection shows and can then be used to model, organize, and manage API requests. It's essentially a container that serves as a bundle of related requests grouped for ease of access, execution, and collaboration. Instead of having API calls scattered across the "History" tab of a browser, collections allow developers to group them logically based on functionality, API version, or any other relevant criteria. Plenty of web app developers have made Postman collections available at https://www.postman.com/explore/collections. Others have them available on a social coding site like GitHub. Many of the tools MacAdmins use to manage devices will have Postman collections, including

- Cisco Meraki: https://www.postman.com/32090/workspace/cisco-meraki-s-public-workspace
- Jamf: https://www.postman.com/grey-zodiac-811335/workspace/team-workspace/collection/25705818-0d4ee2fa-a9cf-4c3d-8fdb-de153e8c5b0a?action=share&creator=3942928
- Kandji: https://www.postman.com/ksapitest/workspace/kandji-api-support-flows/collection/19391022-a556884f-d88a-4bcc-8953-15c15b6810aa?action=share&creator=3942928
- Microsoft: https://www.postman.com/microsoftgraph/workspace/microsoft-graph/overview
- VMware (Carbon Black): https://www.postman.com/carbon-black/workspace/public-carbon-black-workspace/folder/16331452-79e07533-b243-46a8-9afc-690681406ac0

Companies who are serious about making it easy for third-party developers to integrate with them will have a Postman collection.

Collections also have a "Collection Runner" feature that can execute all requests within a collection one after the other, saving time and effort compared to manual runs (and allowing for basic daisy-chaining of actions). Collections can also be paired with

test scripts written in JavaScript, enabling automated validation of API responses for each request. This helps catch regressions and ensures consistent API behavior. Postman can also automatically generate API documentation from collections, including request details, responses, and code snippets. This serves as a valuable reference for developers and users.

Most Postman work is then done in workspaces. This is where a URL can be entered, the method provided, and headers added – which are usually required to provide authentication options and other potentially necessary metadata. To craft the first request, open Postman, click Workspaces, and then click Create Workspace, as seen in Figure 3-8.

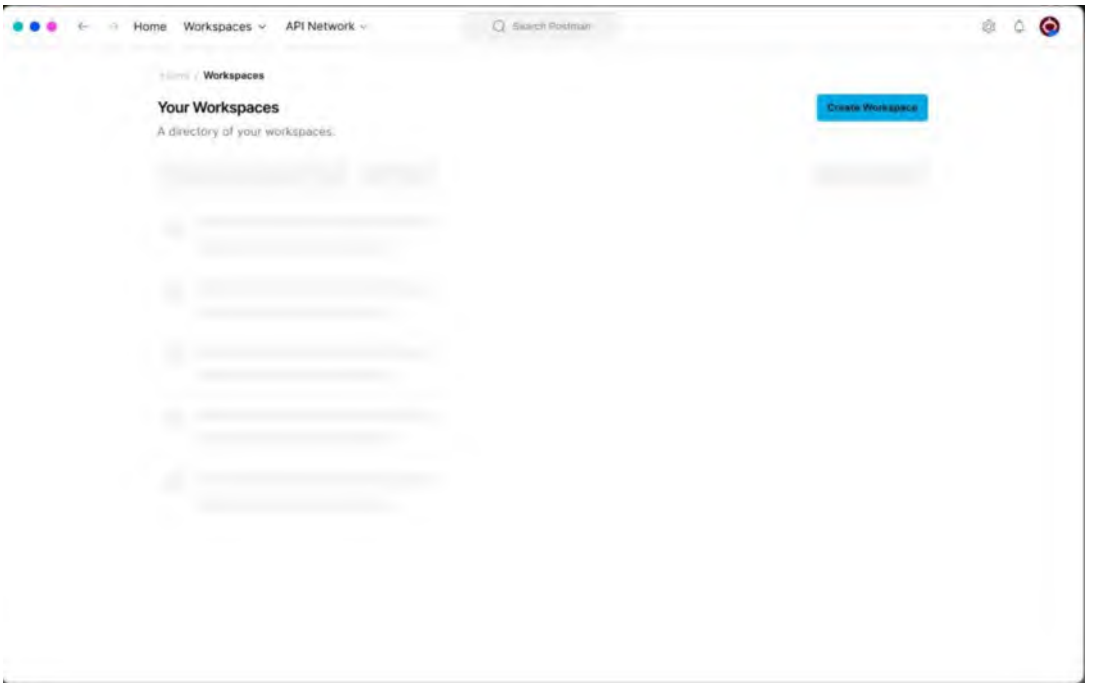

Figure 3-8. *Postman Workspaces*

Click the plus sign in the header to open a new request tab, as seen in Figure 3-9.

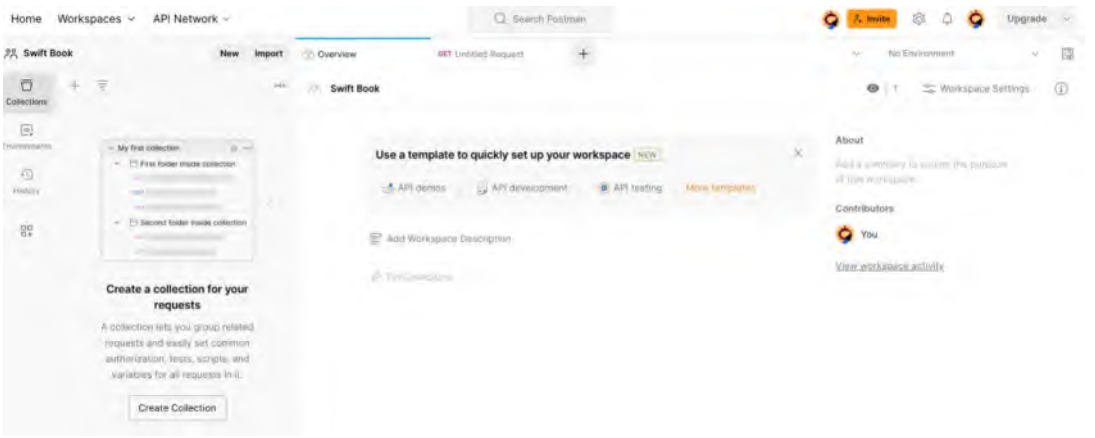

Figure 3-9. *New Request*

At the request screen, specify the request method. Here, select the appropriate HTTP method (GET, POST, PUT, DELETE, etc.) based on the API's endpoint and desired action as seen in Figure 3-10.

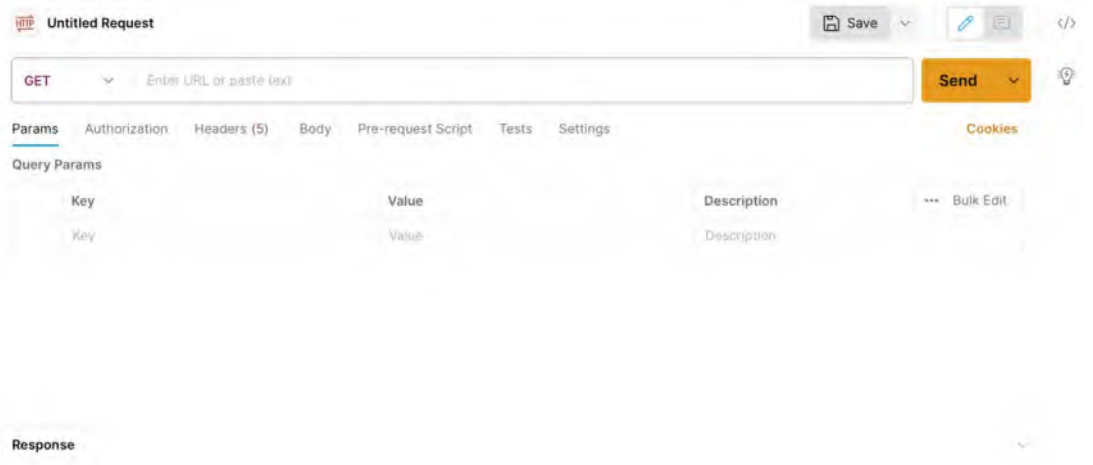

Figure 3-10. *Editing the request*

Provide the URL in the next field over. If the endpoint doesn't require any other information, it would be possible to click the Send button to start the first request at this point. However, most endpoints will require enriching the request at a minimum with a header for authentication. Postman simplifies this by including Authorization as a

tab – but to see what it's doing, click the code icon to see a "Code snippet" panel and then enter a token or OAuth option into the Authorization drop-down and enter the necessary information. As seen in Figure 3-11, the cURL example then reflects the appropriate header entry.

Figure 3-11. *Authorization settings*

Postman also supports API parameters and a number of other features that help craft proper requests. Once the request is written, use the Send button to initiate the request to the API endpoint. Figure 3-12 shows how to get a random fact about a year from the REST API available at `http://numbersapi.com/random/year`.

Figure 3-12. *Sending a request*

Notice the body doesn't show the output in JSON. Use the tab that says Text to switch to JSON, XML, or whatever format is required. The site, like the one used in the example, might also require a parameter be supplied in order to output the results in JSON.

Once good requests are crafted, it's then possible to create collections and even automate tests. JavaScript-based test scripts can be used to automate assertions and validate API responses. Then the Collection Runner can be used to execute multiple requests in a collection sequentially, along with their associated tests if so desired. Users can configure variables for different environments (development, testing, production) to efficiently manage configuration variations as well. Collections and environments can also be shared with others and triggered to automate API testing by incorporating Postman collections into continuous integration and delivery (CI/CD) pipelines to automate testing throughout the development process.

The above entry replied with a simple response. Notice that it shows a status code of 200, as seen in Figure 3-13.

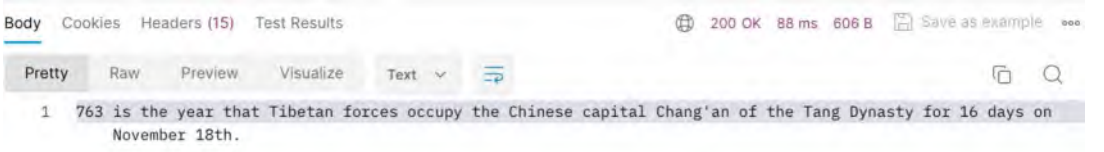

Figure 3-13. *Request response*

This section has been about modeling REST calls so the job goes faster in Swift. No one wants to recompile every time they run a request. There are a number of status codes defined for REST – understanding them and being able to parse them in Swift allows for further telemetry into issues that may be encountered.

Web API Status Codes

When a request is made to a web server, the server will respond with a status code. The status code indicates whether the request was successful or not, and if not, what the error was. There are lots of different HTTP response codes, which are three-digit codes that generally align with the following classes:

- 100-199: Informational responses
- 200-299: Success codes
- 300-399: Redirection responses (further action may be required)

- 400-499: Client errors (although like if a page isn't found on a web server, the server is telling the client that it requested a resource that doesn't exist)
- 500-599: Server errors when trying to process data sent by the client

Of these, some of the most common ones we run into include the following:

- 200 OK: The request succeeded.
- 201 Created: Means that a request was successful and a resource was created.
- 204 No Content: The request succeeded and intentionally returns no response body.
- 205 Reset Content: The server asks the client to reset its document view.
- 206 Partial Content: The server is returning part of the resource (commonly for range requests).
- 300 Multiple Choices: Multiple representations are available for the resource.
- 301 Moved Permanently: The resource has permanently moved to a new URL.
- 404 Not Found: This means that the requested resource was not found.
- 401 Unauthorized: Authentication is required or failed.
- 403 Forbidden: The server understood the request but refuses to authorize it.
- 408 Request Timeout: The server timed out waiting for the request.
- 415 Unsupported Media Type: The request payload's media type is not supported by the server.
- 500 Internal Server Error: This means that there was an error on the server side.

- 503 Service Unavailable: Usually indicates the server is too busy (e.g., needs more resources to process the request).
- 505 HTTP Version Not Supported: Using an HTTP version the server can't interpret.

HTTP response codes are typically more used by web developers (and web browser developers) to better understand the status of a request and to take appropriate action. For example, if a request returns a 404 Not Found status code, the web developer might display a message to the user indicating that the requested resource was not found, like a page with a fail whale. By far, this is the most common code an end user will see, as nearly every web server that listens for requests for a domain has a 404 page – and often customized. Some even break down if the resource being called was deleted or moved and sometimes even a search box to find the right resource. With the amount of pages dynamically generated, it happens often when a butterfly flaps its wings in a category name change on a blog or someone just takes a page down.

Accessing REST Endpoints

The 200 response in the previous section is a standard way to include data from an endpoint. Notice how NoMAD handles response codes when running submitCert:

```
 submitCert(certTemplate: certTemplate, completionHandler: { (data,
response, error) in
     if (response != nil) {
          let httpResponse = response as! HTTPURLResponse
          if (httpResponse.statusCode >= 200 && httpResponse.statusCode
          < 500) {
          }
```

The above example defines acceptable status codes. A request would need to be run to populate the status code, first. This is where URLRequest in Swift comes into play to process the GET request. Swift provides several ways to forge this request. This could be done with direct initialization, as follows:

```
let url = URL(string: "https://api.example.com/users/123")!
var request = URLRequest(url: url)
```

A more easily configurable way, though, would be to use URLComponents:

```
var components = URLComponents()
components.scheme = "https"
components.host = "api.example.com"
components.path = "/users/123"
let url = components.url!
var request = URLRequest(url: url)
```

The request also needs a method, which can be defined as request.httpMethod, set to "GET" as follows:

```
request.httpMethod = "GET"
```

Requests to specific endpoints might require authentication or parameters. So to configure authorization headers, request.setValue can be used to include a Bearer token as an Authorization header, as follows:

```
request.setValue("Bearer YOUR_API_TOKEN", forHTTPHeaderField:
"Authorization")
```

Once these are defined, it's possible to then create the session to go get whatever the endpoint makes available:

```
let session =
URLSession.shared
let task = session.dataTask(with: request) { data, response, error in
if
let error = error {
        print(error.localizedDescription)
    } else
if
let data = data {
        // Parse and handle the response data
    }
}
task.resume()
```

Provided the server responds to the GET request, it's time to parse the response data using built-in mechanisms like JSONDecoder or third-party libraries. Based on the response format, information can easily be extracted and integrated into Swift applications. To access options that help serialize JSON, first import the Foundation libraries:

```
import Foundation
```

Armed with the JSON data, the process of using it typically comes in one of three phases (or a combination thereof):

- Fetch JSON from a remote server using URLSession.
- Load JSON from a local file using FileManager.
- Create JSON data programmatically using string encoding.

To deserialize JSON, use the JSONSerialization.jsonObject(with: options:) to transform JSON data into native Swift types:

```
let jsonData: Data = // Acquire JSON data
do {
    if let json = try JSONSerialization.jsonObject(with: jsonData, options:
[]) as? [String: Any] {
        // Access parsed JSON as a dictionary
    } else if let json = try JSONSerialization.jsonObject(with: jsonData,
options: []) as? [[String: Any]] {
        // Access parsed JSON as an array of dictionaries
    } else {
        // Handle other JSON structures
    }
} catch {
    print("Error parsing JSON: \(error)")
}
```

To access the data, treat parsed JSON as Swift dictionaries and arrays:

```
let name = json["name"] as? String
let items = json["items"] as? [[String: Any]]
```

To handle data types:

- JSON numbers map to Swift's NSNumber type.
- JSON strings map to Swift's String type.
- JSON booleans map to Swift's Bool type.
- JSON null maps to Swift's NSNull type.

Swift objects can then be serialized into JSON as well. Use JSONSerialization.data(withJSONObject: options:) to create JSON data from Swift objects. However, consider optional binding to safely unwrap parsed values and for enhanced type safety and convenience, consider using the `Codable` protocol for custom types. `Codable` is a typealias for `Encodable & Decodable`, and it allows you to encode and decode data from and to JSON.

The following code shows how to parse the JSON data for, let's say, service desk articles:

```
import Foundation
struct Post: Codable {
    let id: Int
    let title: String
    let body: String
}
func parseJSONData() {
    let data = """
        {
            "id": 1,
            "title": "This is a title",
            "body": "This is a body"
        }
    """.data(using: .utf8)!
    let post = try! JSONDecoder().decode(Post.self, from: data)
    print(post)
}
```

There is plenty more that could happen here, like validating parsed data for expected structure and types or using third-party libraries for advanced parsing or performance optimization. It almost might be necessary to chain requests by making subsequent

API calls based on the initial response, building powerful workflows. Given how many apps string together the output from various APIs (be they native to Apple, a REST endpoint, or even some gross old-school websockets API or something like that), it's useful to consider this like some kind of procedural or service-oriented approach to programming.

Figure 3-14 shows a way to visualize that type of approach, using a project open sourced by some of the authors of this book.

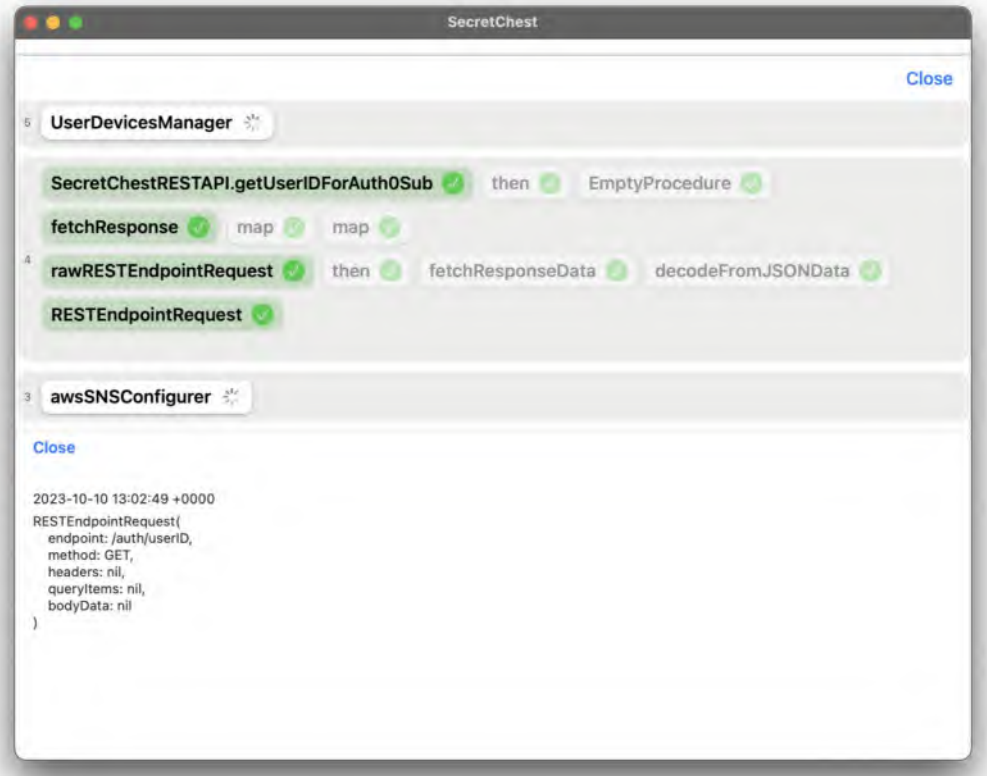

Figure 3-14. *An example app that processes API calls*

Conclusion

The world is just a collection of not-yet interconnected endpoints. At no time has more information been at the fingertips of someone with a decent grasp of how to use a tool like Postman or how to wire up some APIs from operating system manufacturers to public (or even private) APIs. There are more packages, pods, and any other type of integration point than anyone can keep up with, but the patterns repeat.

CHAPTER 4

Authentication and Authorization

Authentication and authorization are two important concepts in security. Authentication is the process of verifying the identity of a user or device, while authorization is the process of determining what actions a user or device is allowed to take.

Authentication is typically done by requiring the user or device to provide some form of identification, such as a username, password, security token, or all of the above. There are two main types of authentication: something you know and something you have. Something you know authentication methods typically involve the user providing a username and password. Something you have authentication methods typically involve the user providing a physical token, such as a security key or a smart card.

Authorization is the process of determining what actions a user or device is allowed to take. This is typically done by assigning the user or device to a role or group. The role or group will then have a set of permissions associated with it, which determine what actions the user or device is allowed to take. This is usually referred to as "RBAC" or Role Based Access Control. Other methods of determining authorization include Attribute Based Access Control, where an attribute of a user, perhaps geographic location, who the user's manager is, or even what time the operation is happening.

The main difference between authentication and authorization is that authentication is about verifying the identity of a user or device, while authorization is about determining what actions a user or device is allowed to take. For example, a user who is assigned the role of "administrator" may be allowed to create new users, modify user permissions, and access sensitive data. A user who is assigned the role of "user" may only be allowed to create and modify their own data.

Authentication is typically the first step in the authorization process. Once a user or device has been authenticated, the authorization process can then determine what actions the user or device is allowed to take.

C. Edge et al., *Beyond Scripting*, https://doi.org/10.1007/979-8-8688-3035-8_4

The reason we look at authentication and authorization separately is that the technology that facilitates each is different. There may be elements of authorization in an authentication request, but for the most part, the authentication is done using open standards, and authorization is a way of handling permissions within a given app or web app. Because authentication happens first, let's look at that first.

This chapter covers

1. Core concepts (authentication vs. authorization)
2. Transport security (TLS/SSL) for trusted communication
3. Token-based authentication and authorization (JWT, bearer tokens, OAuth)
4. Identity providers and federation (Login with Apple/Facebook/Google, OIDC, SAML)
5. Modern authentication methods such as passkeys

Authenticating to Web Services

Once upon a time, an app, or "fat client" as they were often referred to, had to connect to a socket on some server. That might stream bits in strange and unpredictable ways. Admins had to know the ports that needed to be open on firewalls, for ingress and egress traffic, in order to allow for connectivity. It could be a mess, with each vendor often doing things their own way.

Today, most apps tend to connect to a REST endpoint on some web server in order to exchange data. Rather than stream bits that get interpreted by a custom listener, apps tend to exchange data with those endpoints with standard markup languages like XML, JSON, or YAML – or for old-school Apple apps, dictionaries. Most apps also don't require users to log in for every session they open. Users don't provide a username and password to those endpoints every time data needs to be exchanged. Instead, developers have users enter credentials and then escrow a key that provides for subsequent communications. Those keys, or tokens, usually come in the form of a token, a token wrapped up in a markup language, or a more modern approach like OAuth. All of these are made possible by an SSL-backed connection to a server, which uses a public and private keypair to verify that a device is authentic and that the traffic to and from the device doesn't have any other devices listening to and playing back traffic.

SSL

SSL/TLS is a cryptographic protocol that safeguards communication by encrypting data and verifying identities. Certificates are digital documents that bind a public key to an entity (e.g., server, device), enabling authentication and encryption. Trust stores are collections of trusted certificates used to verify server identities. Client-side certificates are client identities (certificate + private key) used by clients to authenticate themselves to servers. This is fairly transparent on most Apple devices because Keychain, and the operating system itself, handle most of the heavy lifting.

Security Note: Always use HTTPS/TLS for authentication and authorization traffic. Do not transmit credentials or tokens over plaintext HTTP.

Apple actually makes it pretty hard not to use secure methods to load remote content. Using URLSession and other built-in methods with Swift will force you to use proper security hygiene while also doing most of the work for you. It's only if you need to do something well off the beaten path that you should need to alter the default behaviors.

Integrating SSL/TLS into Swift projects to handle authentication, device verification, and server authorization typically relies on the device's built-in trust configuration for verifying server certificates. For custom trust stores, use the SecTrustSetAnchorCertificates function. To create secure connections, use URLSession's dataTask(with:) for HTTPS requests and URLSessionConfiguration's tlsMinimumSupportedProtocolVersion property to require a minimum TLS version. For example, consider the following code, which establishes a secure HTTPS connection to the specified server using SSL/TLS (defined as URL()) and fetches data from the server:

```
let url = URL(string: "https://secure.example.com")!
let session =
URLSession.shared
let task = session.dataTask(with: url) { data, response, error in
if
let error = error {
        print("Error: \(error)")
        return
    }
    // Handle secure response data
}
task.resume()
```

The let url line constructs a URL object representing the address "`https://secure.example.com`".

The ! operator forcibly unwraps the optional result, indicating confidence in the URL's validity.

The let session line then retrieves a shared URLSession, the primary class for managing network data tasks. The shared session is a convenient way to perform common network requests without extensive configuration. The let task then creates a data task associated with the specified URL. The dataTask is a type of network request that retrieves data from a server.

The closure provided after dataTask(with:) is executed upon task completion or failure.

The let error handles errors and task completion. If an error occurs, it's printed to the console, and the closure exits. The "// Handle secure response data" is a placeholder comment, intended for processing the received data if the request is successful. The actual implementation for handling the data would be placed here, depending on the specific application requirements. The task is then started with the last line that includes task.resume(). This line initiates the execution of the data task, sending the network request to the server.

When communication fails, it helps to have tools like openssl on a Mac. For example, for a web server an admin might traditionally attempt to telnet into port 80 on and check your banners, if the goal is to get a certificate, over port 443, the openssl command will be more handy. The following example uses the openssl command from a Terminal session to tell openssl to be a generic client (s_client) and connect (-connect) to `https://krypted.com/` over port 443:

```
openssl s_client -connect krypted.com:443
```

The output should then look similar to the following:

```
CONNECTED(00000003)
depth=3 /L=ValiCert Validation Network/O=ValiCert, Inc./OU=ValiCert Class 2
Policy Validation Authority/CN=http://CERTAUTHORITY.com//
```

Test smtp using the same, whether using port 25 and requiring a certificate or another port. To test with port 25, assuming a generic client again and changing the port number and because SSL can work with smtp directly, using starttls to do so:

```
openssl s_client -connect  www.krypted.com:25 -starttls smtp
```

A valid connection would result in similar output to the following:

```
CONNECTED(00000003)
depth=3 /L=ValiCert Validation Network/O=ValiCert, Inc./OU=ValiCert
Class 2 Policy Validation Authority/CN=http://MYCERTAUTHORITY.com//
emailAddress=krypted@mac.com
```

It's also possible to initiate a new instance of an SSL listener, using s_server or just test the connection timer using s_time. This can help troubleshoot issues in the swift code that will be used in the next couple of sections, when needed. Overall, openssl is a pretty invaluable toolkit that we'll probably look at more and more on this site.

If a server issues certificates, those certificates are validated during TLS negotiation; they are not automatically added to Keychain unless explicitly installed. Keychain can also store other types of tokens. We'll get to what a bearer token is later, but for now, consider the following code, which sends a bearer token over HTTPS:

```
let url = URL(string: "https://server.example.com")!
let configuration =
URLSessionConfiguration.default
configuration.httpAdditionalHeaders = ["Authorization": "Bearer <token>"]
let session =
URLSession(configuration: configuration)
let task = session.dataTask(with: url) { data, response, error in
    // ...
}
task.resume()
```

Notice that the Authorization is a Bearer Token. Those are acquired through authentication services or API calls and then included in the Authorization header of HTTPS requests. Keep in mind, more advanced techniques, are available. Certificate Pinning is used to restrict acceptable certificates to a specific set for enhanced security against Man in the Middle (or MITM for short) attacks. Also, secure networking libraries like Alamofire for easier SSL/TLS handling and advanced features. Certificate-based workflows like SCEP/ACME can coexist with token-based authentication and authorization workflows.

Automating Certificate Management with SCEP and ACME

Many modern apps use certificates, and some with an eye towards security will automate certificate enrollment and the full lifecycle of certificate management. Given that the two main mechanisms to do so are SCEP and ACME, it's worth comparing SCEP and ACME Certificate Management for swift to secure communication between applications and external entities

SCEP has been a trusty companion for certificate enrollment for years (it originated in the 1990s and was later standardized as RFC 8894). Its workflow is straightforward: an application interacts with a dedicated SCEP server, authenticates, and requests a signed certificate based on predefined settings. Upon successful validation, the server issues a certificate that's unique to that device (rather than a public key shared by all devices that interact with a web server). The app then installs and uses its unique certificate for secure communication. Given the standardization and how long it's been around, there are plenty of CAs and servers with built-in SCEP functionality.

Swift libraries like "Alamofire" can facilitate HTTPS communication, and administrators can configure SCEP servers to enforce specific certificate policies.

SCEP worked great for 20+ years, but ACME is more mobile friendly. ACME offers a dynamic and automated approach to certificate management. Applications assume an active role, directly interacting with an ACME server to request and manage certificates. Certificates are then provisioned on-demand, responding to an application's specific needs dynamically.

Public-key cryptography replaces shared secrets, which minimizes vulnerabilities, and can be combined with Apple attestation mechanisms in architectures that require device trust signals.

For established infrastructure with static certificate requirements, SCEP offers a familiar and efficient solution. For more automation, security, and mobile-friendliness, ACME's dynamic approach provides a future-proof option. There are also several tools that simplify SCEP and ACME integration, often handled by backend infrastructure, MDM tooling, or platform-specific APIs. It's also useful to understand the flow of each, in order to best ascertain which works better for a given use case.

The SCEP Flow

The SCEP flow involves defining the desired certificate settings within an app, including key size, algorithm, and subject information. Enrollment in the SCEP service happens when the app connects to a designated SCEP server, typically using HTTPS for secure communication. Authentication then uses the app's credentials (e.g., username/password or certificate) to authenticate with the server.

Once authenticated, a SCEP request message is sent, containing the configured settings and authentication information. The server validates the request and potentially verifies additional information like device identifiers where appropriate. Upon successful validation, the server issues a signed certificate for the app. The app receives and installs the certificate, ready for secure communication.

To get SCEP integrated into a Swift app, first choose a SCEP-capable client or service layer that can handle communication with the SCEP server, simplifying the process. The implementation then needs to be integrated into the Xcode project. Then configure it, specifying the server URL, the app's credentials, and desired certificate settings. Then trigger the enrollment function to send the SCEP request. This involves handling the response by processing the server's response, checking for successful certificate issuance, and/or isolating any potential errors. If all that is successful, use platform-specific APIs to install or reference the certificate identity in Keychain. Now let's look at all this as it appears in code:

```
// Define settings and server URL
let settings = YourSCEPSettings(
    keySize: 2048,
    algorithm: "rsa",
    subject: "CN=device.example.com"
)
let serverURL = URL(string: "https://scep.example.com")!
// Create SCEP client and configure
let manager = YourSCEPClient(url: serverURL)
manager.settings = settings
manager.credentials = ("user", "password")
// Initiate enrollment
manager.enroll { result in
    switch result {
```

```
    case .success(let identity):
        // Install and use the certificate
        print("Certificate successfully enrolled!")
    case .failure(let error):
        // Handle error
        print("Error enrolling certificate: \(error)")
    }
}
```

To unpack what the above code does, it first defines settings for the SCEP server, including key size, algorithm, and subject. Then it defines the URL to the server (`serverURL`). The script then uses `manager` to create a SCEP client instance for managing communication with the SCEP server. The `manager.settings = settings` sets configuration settings for the certificate request and `manager.credentials = ("user", "password")` specifies credentials to authenticate with the SCEP server.

Once the environment and configuration is set, the manager.enroll initiates the enrollment, which triggers the certificate enrollment process, sending a request to the SCEP server.

The closure handles the result of the enrollment, either success or failure. Inside the closure is a success case, so if enrollment is successful, the server issues a certificate and the script prints "Certificate successfully enrolled!" Otherwise, there's a failure case, that gets captured and printed to the console. The actual implementation for installing and using the certificate is left as a placeholder, requiring further context for completion – given that every app is different.

Again, though, SCEP isn't the only option. As mentioned, ACME has a few benefits over SCEP, so let's look at how the ACME flow works.

The ACME Flow

The ACME workflow isn't completely unique, now that the SCEP flow was covered in the previous section. An app registers with an ACME server, obtaining an account identifier and initial challenge. The server then presents a challenge to prove control of a domain (or other supported identifier). This can involve responding to challenge types such as HTTP-01 or DNS-01 to provide that verification. Once the challenge is successfully completed, the app sends a signing order request specifying the desired certificate settings. Upon validation, the ACME server issues and signs the certificate for your app. The app then receives and installs the certificate, now ready for secure communication.

Integrating ACME, as with most other authentication mechanisms, then begins with selecting an ACME client implementation. Once selected, it will be necessary to configure the client. Here, provide it with the server URL, desired account identifier, and certificate settings. Once done, it's then possible to initiate registration, triggering the registration function to acquire an account and initial challenge. The challenge will then need to be solved, with the code implementing logic to respond to the specific challenge presented by the server, utilizing cryptographic libraries if needed.

Once the challenge is resolved, use the library to send a signing order request specifying the desired certificate settings. Then employ the library's polling mechanism to check for the issued certificate from the ACME server and finally install the certificate. Upon receiving the certificate, install it in the certificate store for the endpoint that will terminate TLS (e.g., a server, reverse proxy, or managed endpoint). Let's look at all this in action:

```
// Define server URL and certificate settings
let serverURL = URL(string: "https://acme-v02.api.letsencrypt.org")!
let settings = YourACMESettings(
    domains: ["yourdomain.com"],
    keySize: 2048,
    algorithm: "rsa"
)
// Create ACME manager and configure
let manager = YourACMEClient(url: serverURL)
manager.settings = settings
// Initiate registration
manager.register { result in
    switch result {
    case .success(let account):
        // Proceed with challenge-response and certificate issuance
    case .failure(let error):
        // Handle error
    }
}
```

Let's unpack what this code does. As with the SCEP example, it sets `serverURL` as the URL of the ACME server. It then creates ACME settings with the key size, algorithm, and domain to use and pulls this into `manager` as an ACME client. The `manager.settings` then sets the configuration settings for the certificate request and `manager.register` initiates the actual registration.

The registration process with the ACME server is used to obtain an account and initiate the certificate issuance process. Then the closure handles the result of the registration, either success or failure. The success case creates an account on the ACME server, and the failure case gets output to console. The actual implementation for challenge-response and certificate installation is left as a placeholder, requiring further context for completion, again, given that every app is different.

Certificates, or their automated services like SCEP and ACME, represent a way to authenticate an app. They can be used for a variety of tasks. But while these can be issued based on user accounts, consider them device-based authentication. Client identities are typically stored in Keychain. A user can also authenticate and get a token, or authenticate for every session (or more likely some kind of combination thereof).

JWT, Bearer Tokens, and OAuth Oh My

In the vast landscape of API security, navigating through terms like JWT, bearer tokens, and OAuth can feel like cracking a cryptic code. They are similar:

- *JWT*: The JSON Web Token (JWT) is a self-contained, signed piece of information encoded in JSON format. It carries data about a user (claims) – like username, role, or permissions – and is verified by a cryptographic signature. Think of it as a secure passport that grants access to specific resources without revealing sensitive details.
- *Bearer Token*: Unlike JWT, a bearer token is simply a string used for authorization. It doesn't hold any user information directly. Instead, it serves as a placeholder that the API can verify against a separate system (like an authorization server) to confirm the user's right to access resources. Imagine it as a key card that unlocks doors without a developer needing to show an ID every time.

- *OAuth*: Now, OAuth isn't a token itself, but rather a standardized protocol for authorization. It dictates how different parties – clients, users, and authorization servers – interact to grant access to protected resources. Think of it as the intricate choreography of a security tango, where each party plays a specific role according to the defined rules.

This can get confusing as a "JWT with Bearer Token" authentication type means that JWTs can be used as bearer tokens. The API checks the token signature to verify the user's claims and grant access. "OAuth with Bearer Token" means that OAuth is commonly used to issue bearer tokens. After users successfully go through the OAuth flow, they receive a token used for subsequent API calls. To break this down, JWTs are often used for stateless APIs, simple authentication, and sharing user information securely. Bearer tokens are best for stateless APIs and separating tokens from user data. OAuth should be used for complex authorization flows, integrating with third-party providers, and managing access scopes. Or actually, default to OAuth and only use one of the others if it's not available.

Security Note: Never log tokens or credentials in application logs, debug output, crash reports, or analytics payloads.

Once developers have decided which to use, it's time to look at how each appears in code.

Using a JWT

JSON Web Tokens (JWTs) offer a secure and versatile way to send and verify user information between applications. Handling JWTs in swift is surprisingly straightforward with available libraries and built-in tools. The first step, as it will be with most every aspect of integrating authentication options into code, is to choose a library. Popular options include JWTDecode and Swift-JWT. JWTDecode is commonly used for parsing claims, and Swift-JWT is commonly used for signing and verification.

Once selected, the first step is to load the token: Pass the JWT string to the library's decoding function. To access the JWT data, decode the header and claims into objects. Claims are dictionaries containing user information like username, ID, and roles. For example, to do this with JWTDecode:

```
import JWTDecode

let token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
let jwt = try decode(jwt: token)
```

```
if let username = jwt.claim(name: "username").string {
    print("Welcome, \(username)!")
} else {
    print("Invalid token!")
}
```

Note An actual token is much longer than the one in the above script.

To unpack what the above code does, it begins by defining a token (as a constant). The decode function then attempts to parse the token. The `jwt` value then exposes claims such as username. Finally, a properly unpacked token prints a welcome to the user or prints that it's invalid.

Note Keep in mind that the above code assumes the JWTDecode library is imported and available and that the appropriate claims are present. Also, decoding alone does not verify the signature.

Encoding a JWT involves creating a signed identity. To do so, first prepare claims. This is a dictionary containing the user information to include in the token. To do so, choose a signing algorithm like HS256 or RS256. This algorithm combines your secret key with the claims to create a unique signature. Then use the library's encoding function, supplying the header, claims, secret key, and algorithm. Once created, treat the encoded JWT as sensitive information and store it securely on the client or server, depending on the use case.

Here's an example using Swift-JWT:

```
import Foundation
import SwiftJWT

struct MyClaims: Claims {
    let username: String
    let role: String
}
```

```
let claims = MyClaims(username: "johndoe", role: "admin")
var jwt = JWT(claims: claims)
let signer = JWTSigner.hs256(key: Data("supersecret".utf8))
let token = try jwt.sign(using: signer)
print("Your encoded JWT: \(token)")
```

There are a few more things to consider when using JWTs. They can have expiration times. Use library functions or manually check the "exp" claim to ensure token validity. Implement refresh tokens for long-lived sessions. These allow acquiring new access tokens without requiring user re-login (because let's face it, users LOVE to log into management tools so they can do more of the things to their devices).

It's also possible to use custom claims, which can be used to store additional user data specific to a given application. Again, though, in the onslaught of options available for authentication and authorization, there's also bearer tokens to consider.

Using a Bearer Token

Remember, bearer tokens are commonly used in RESTful APIs for authorization after authentication, to verify if the user is logged in and authorized to access specific resources. They can also be used for API Throttling or to limit the number of requests a user can make based on their token permissions. Bearer tokens are also useful for resource access control or to control access to specific API endpoints based on token roles and privileges.

Bearer tokens can be used for login and for an authorization flow. An app might obtain the token through a user login process or authentication service integration. Some APIs might also return the token as part of the response to a specific request. Once obtained, tokens should be stored securely. Never store the token directly in your code or embed it in your app. Use secure options like the iOS Keychain or dedicated library for encrypted storage. Make sure that authorized parts of an app can then access the token during API calls.

To add the authorization header, first create a URLRequest object and add the "Authorization" header with the value "Bearer <your_token>". Many libraries like Alamofire and URLSession offer built-in functionalities for adding authorization headers based on a provided token. To then make API calls, first send the request, which is sent as the prepared URLRequest with the bearer token header to the desired API endpoint. Then analyze the response to assess successful access or potential errors related to the token's validity or permissions.

Before getting in to sample code, remember to only use bearer tokens with HTTPS connections to ensure encrypted communication. Avoid any unnecessary exposure of the token in logs or debugging statements and make sure to implement token refresh mechanisms for long-lived sessions to avoid relying on a single token indefinitely. Consider shorter expiry times for sensitive APIs to minimize potential damage from compromised tokens.

Keeping all this in mind, let's look at an example use of a bearer token:

```
let url = URL(string: "https://api.example.com/endpoint")!
var request = URLRequest(url: url)
request.httpMethod = "GET"
// Load token from secure storage
let token = KeychainWrapper.shared.getToken()
request.setValue("Bearer \(token)", forHTTPHeaderField: "Authorization")
// Perform API call using your preferred networking library
```

The above code first creates the URL, as has been done with each of the samples in the chapter thus far. This involves populating the url as a string to an endpoint. The ! operator forcibly unwraps the optional result, indicating confidence in the URL's validity. The URLRequest initializes a URLRequest object, which encapsulates the details of an HTTP request. The method is then defined in request.httpMethod as a GET request, which specifies that the request will use the HTTP GET method, commonly used to retrieve data from a server.

The next part of the code defines that the token is retrieved from Keychain, in this instance as KeychainWrapper.shared.getToken() assumes a helper class or library called KeychainWrapper for secure token storage. The request.setValue() then appends an "Authorization" header to the request, containing the retrieved token prefixed with "Bearer".

This header is often used in token-based authentication schemes like OAuth as well.

Finally, the commented out final line is a placeholder for API call, which indicates that the next step involves executing the prepared request using a networking library like URLSession or another from Alamofire. Thus far, the chapter has dealt with traditional means of accessing tokens in the same way websites have done for over a decade. But Apple decided to go a step further and provide their own mechanism with more of an emphasis on privacy protection (which is covered later in the chapter).

Summary checkpoint: The sections up to this point covered transport security (TLS), certificate-based trust, and token-based patterns (JWT, bearer tokens, OAuth) that underpin most modern app authentication flows.

Login with Apple

One of the most common services that users will want to authenticate with for Apple environments is Login with Apple. Most admins will know all too well exactly what this is and how it works, so this book doesn't have to belabor the point. Having said that, if the point needs to be drilled in, consider the following link for more information: `https://support.apple.com/en-us/HT204053`.

To use Login with Apple, first enable the capability in the app. To do so, open an Xcode project and select an app target, go to the "Signing & Capabilities" tab, and click the "+" button and search for "Sign In with Apple" (Figure 4-1).

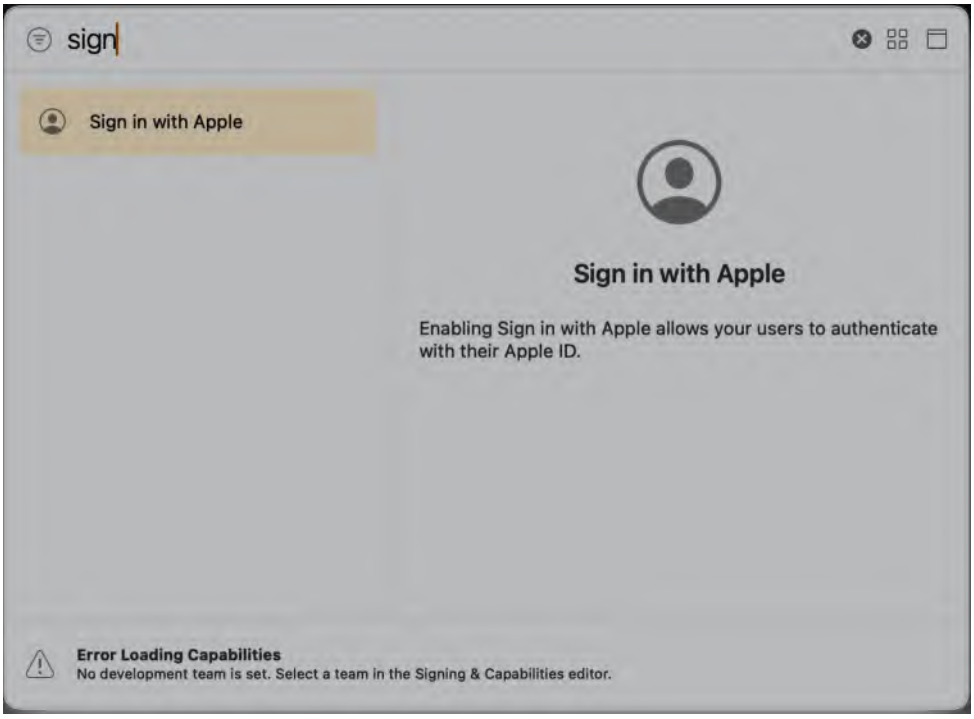

Figure 4-1. *Adding Sign in with Apple to an app in Xcode*

Double-click the capability to add it. To then initialize the authorization provider, import the AuthenticationServices framework. Create an instance of ASAuthorizationAppleIDProvider and configure the request, using the provider to create an ASAuthorizationAppleIDRequest. Then specify the requested scopes: .fullName, .email, etc., and optionally, configure options like private email and contact disclosure.

To build the sign-in button, add an ASAuthorizationAppleIDButton to your interface and wire the button tap action, selecting the desired button style (black, white, outline) in the process. Handle the button tap itself as laid out in Chapter 1. Implement the ASAuthorizationControllerDelegate method authorizationController(controller:didCompleteWithAuthorization:) and get the ASAuthorization object and its credential. Extract user information like user ID, name, and email (if the account used is authorized for such lofty information). Use the user ID to identify and register the user in your system.

The authorizationController(controller:didCompleteWithError:) delegate method can then be used to handle potential errors like user cancellation or network issues. There are plenty of resources that outline exactly how to use Login with Apple in Swift and offer nearly perfect example code, like https://developer.apple.com/documentation/authenticationservices/implementing_user_authentication_with_sign_in_with_apple.

These also show how to personalize the experience with custom UI and animations, configure Apple ID credential options like private email and iCloud Keychain integration, and even implement Apple's Single Sign-On (SSO) feature for seamless login across platforms. Another option that's fairly well documented around the web given how long it's been around is the Login with Facebook option so common with apps, probably far more common with apps that target consumers (although some admins might find it useful, so it's here in this chapter).

Facebook Login

Adding the ability to login to Facebook can make it easier to get started with a service that requires some form of login. This can save people time, provided they have a Facebook account (and most probably do at this point, even if they don't really use Facebook for much more). This will require a Facebook developer account, so the first step is to create a Facebook app at https://developers.facebook.com/ using the "Create App" button. Developers will need to provide some basic information about the app, such as its name and purpose. Once created, the developer account will get a set of credentials required to integrate Facebook into an app.

The steps can change over time, so check with https://developers.facebook.com/docs/facebook-login/ios/ to make sure all requirements are met and that the process is still accurate, given that the following steps borrow the code samples from that page. In the meantime, the next step is to add the Facebook SDK to the app. This is done by adding the package from https://github.com/facebook/facebook-ios-sdk to the Swift Package Manager (SPM).

Next, configure the Info.plist file with an XML snippet that contains data about the app.

Once Facebook Login is enabled, certain App Events are automatically logged and collected for Events Manager, unless Automatic App Event Logging is disabled. To do so, right-click the Info.plist file in the app and select Source Code from the Open As menu. This allows for pasting raw XML into the file in order to configure the CFBundleURLScheme and the necessary keys to make the various options work. The following will be pasted inside a dictionary:

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
  <key>CFBundleURLSchemes</key>
  <array>
    <string>fbAPP-ID</string>
  </array>
  </dict>
</array>
<key>FacebookAppID</key>
<string>APP-ID</string>
<key>FacebookClientToken</key>
<string>CLIENT-TOKEN</string>
<key>FacebookDisplayName</key>
<string>APP-NAME</string>
<key>LSApplicationQueriesSchemes</key>
<array>
  <string>fbapi</string>
  <string>fb-messenger-share-api</string>
</array>
```

Next, replace APP-ID with the App ID supplied when the app was registered on the Facebook developer portal. Replace the APP-ID in FacebookAppID with the App ID as well. Replace CLIENT-TOKEN in FacebookClientToken with the Client Token in the App Dashboard on the Facebook developer portal. Replace APP-NAME in FacebookDisplayName with the name of the app.

The Keychain is used to store tokens for Facebook, as with other login types covered. Therefore, the Keychain Sharing capability is only needed for apps that share Keychain items across app targets/extensions (older Catalyst scenarios can require this). From Xcode, go to the Signing & Capabilities section for the project and click the + Capability button to configure this option for the app target (Figure 4-2).

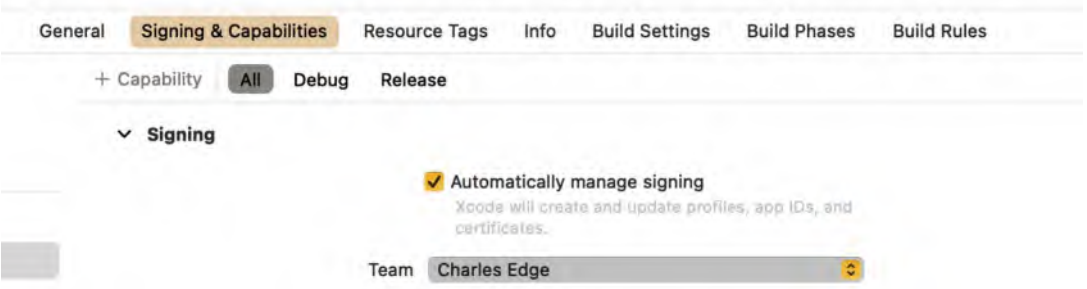

Figure 4-2. *Signing & Capabilities for an app in Xcode*

Search for the Keychain Sharing capability and double-click it to add it to the project, as in Figure 4-3.

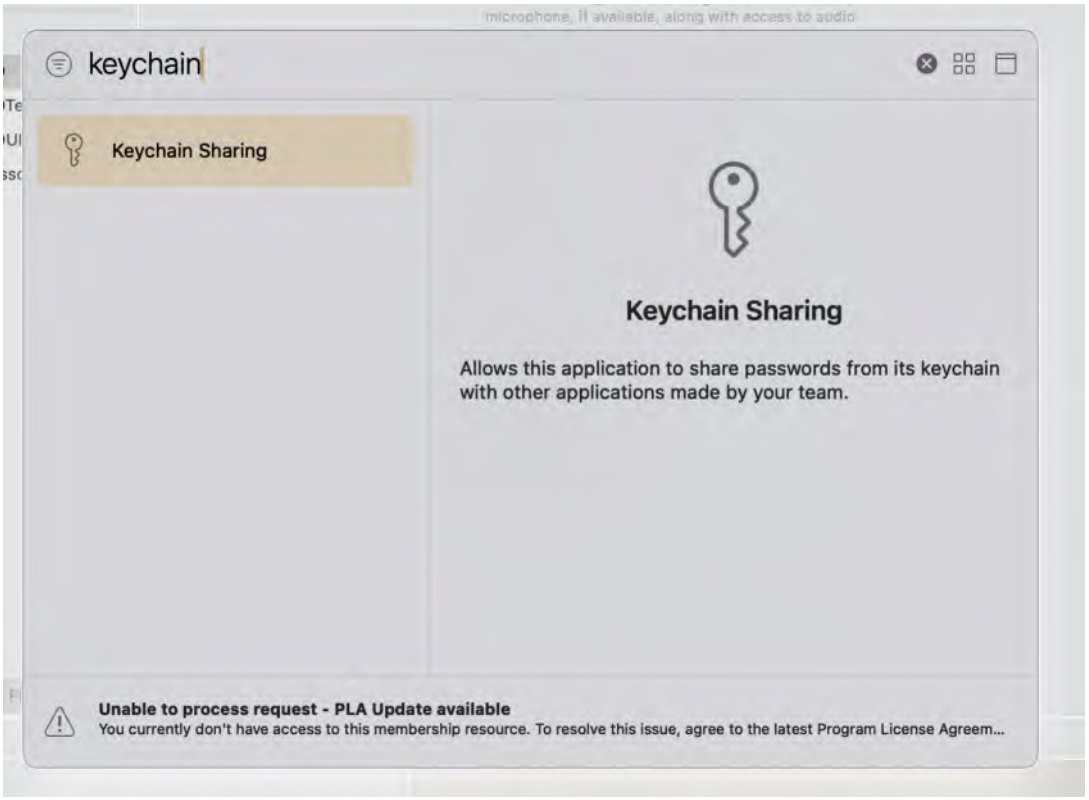

Figure 4-3. *Adding Keychain Sharing to an app in Xcode*

If the Keychain Sharing was added properly, it will appear in the list of capabilities, as seen in Figure 4-4. If the app has multiple targets, groups will also needed to be added.

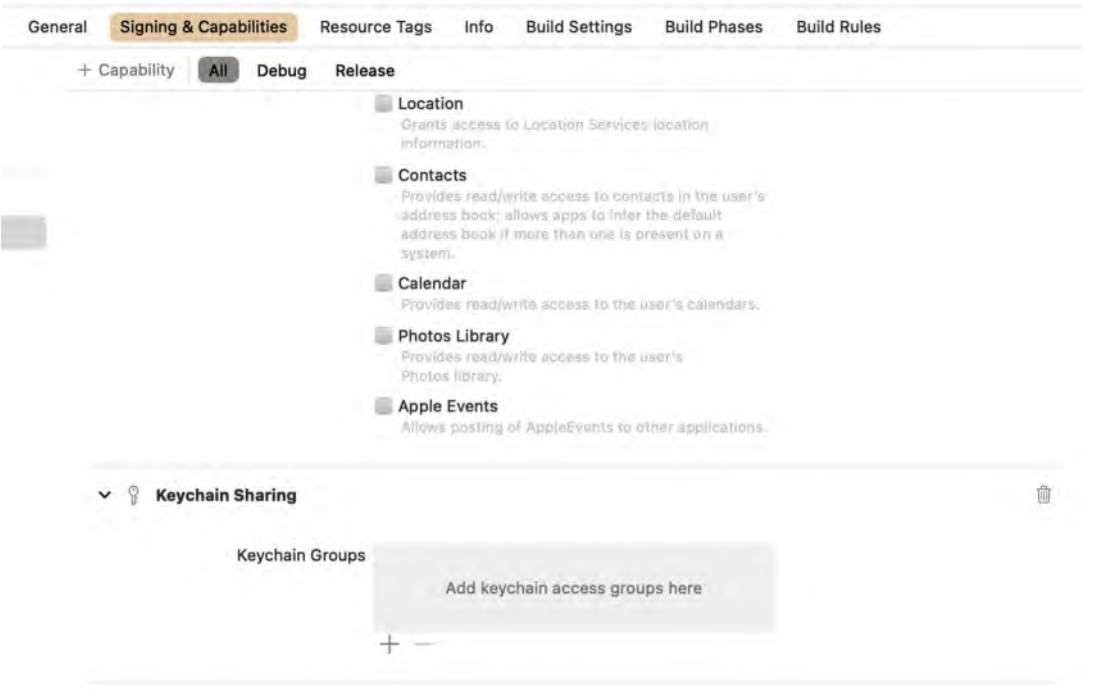

Figure 4-4. *Keychain Sharing options after it has been added*

Now it's time to get to the code examples (thank you to Facebook for making this so easy). To quote their developer page:

Replace the code in AppDelegate.swift method with the following code. This code initializes the SDK when your app launches, and allows the SDK to handle logins and sharing from the native Facebook app when you perform a Login or Share action. Otherwise, the user must be logged into Facebook to use the in-app browser to login.

```
// AppDelegate.swift
import UIKit
import FacebookCore
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [UIApplication.
        LaunchOptionsKey: Any]?
```

```
    ) -> Bool {
        ApplicationDelegate.shared.application(
            application,
            didFinishLaunchingWithOptions: launchOptions
        )
        return true
    }
    func application(
        _ app: UIApplication,
        open url: URL,
        options: [UIApplication.OpenURLOptionsKey : Any] = [:]
    ) -> Bool {
        ApplicationDelegate.shared.application(
            app,
            open: url,
            sourceApplication: options[UIApplication.OpenURLOptionsKey.
            sourceApplication] as? String,
            annotation: options[UIApplication.OpenURLOptionsKey.annotation]
        )
    }
}
```

Next, add a method to the SceneDelegate:

```
// SceneDelegate.swift
import FacebookCore
  ...
func scene(_ scene: UIScene, openURLContexts URLContexts:
Set<UIOpenURLContext>) {
    guard let url = URLContexts.first?.url else {
        return
    }
    ApplicationDelegate.shared.application(
        UIApplication.shared,
        open: url,
```

```
        sourceApplication: nil,
        annotation: [UIApplication.OpenURLOptionsKey.annotation]
    )
}
```

To then the actual button, the following gets added to the ViewController:

```
// Add this to the header of your file, e.g. in ViewController.swift
import FacebookLogin
// Add this to the body
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let loginButton = FBLoginButton()
        loginButton.center = view.center
        view.addSubview(loginButton)
    }
}
```

Each person who logs in then gets an AccessToken.current, stored in the Keychain, which comes with a userID – so make sure there's logic to separate data seperately, if any is stored at rest, if appropriate. Also check that there's an existing token to keep from showing the login flow again, if appropriate:

```
override func viewDidLoad() {
    super.viewDidLoad()
    if let token = AccessToken.current,
        !token.isExpired {
        // User is logged in, do work such as go to next view controller.
    }
}
```

Finally, the app needs to ask a user for their permission to see certain aspects of data on the device. To quote Facebook:

When using Facebook Login, your app can ask for permissions on a subset of a person's data. Facebook Login requires advanced public_profile permission, to be used by external users.

The request will be made to the user when the permissions property on the FBLoginButton object is first used:

```
// Extend the code sample from 6a. Add Facebook Login to Your Code
// Add to your viewDidLoad method:
loginButton.permissions = ["public_profile", "email"]
```

The user is then prompted when this stanza is first encountered. And that's pretty much it. Now there's plenty of logic to consider like how to deal with multiple users, or if this is for an internal company app, how to know what Facebook users match to what other user objects at company. Or if it's a direct to consumer rather than internal app, now users can log in.

Login with Google

Login with Google is another popular means of logging in users, especially if an organization uses Google Workspace for their identities. The first step to integrate a "Login with Google" button is to create a Google Developer account. This is done at the Google Developers website: https://console.developers.google.com/ using the "Create Project" button. Fill in some basic information about the project. Once created, this provides a set of credentials that is required to integrate Google into an app.

Note Google provides the official package at https://github.com/google/GoogleSignIn-iOS. This section uses the package regardless of whether it is installed with Swift Package Manager or CocoaPods.

The next step is to add the Google Sign-In SDK to the app. This can be done by adding the following line to an app's Podfile:

```
pod 'GoogleSignIn'
```

Once added, the Google Sign-In SDK will need to be imported into the code. Do this by adding a statement to import GoogleSignIn to the top of the code as follows:

```
import GoogleSignIn
```

The final step is to implement the login flow. This involves creating a button that users can tap to login to the app with their Google credentials. When the user clicks the button, the Google Sign-In SDK will open a dialog box where the user can enter their Google credentials. If the user enters their credentials correctly, they will be logged in to your app.

Here is an example of how to implement the login flow in Swift:

```
import GoogleSignIn
class ViewController: UIViewController {
    @IBAction func loginButtonTapped(_ sender: Any) {
        GIDSignIn.sharedInstance.configuration = GIDConfiguration(clientID:
        "YOUR_IOS_CLIENT_ID")
        GIDSignIn.sharedInstance.signIn(withPresenting: self) { result,
        error in
            if let error = error {
                print(error)
            } else if let user = result?.user {
                let userId = user.userID ?? ""
                let email = user.profile?.email ?? ""
                print("User ID: \(userId)")
                print("Email: \(email)")
            }
        }
    }
}
```

This code creates a button that, when clicked, will open a dialog box where the user can enter their Google credentials. If the user enters their credentials correctly, their email address will be printed to the console. To break it down:

import GoogleSignIn: This line imports the GoogleSignIn library, providing tools for integrating Google Sign-In functionality into your app.

The ViewController class for UIViewController declares a view controller class, responsible for managing the user interface. The @IBAction func loginButtonTapped(_ sender: Any) connects a button in the user interface to this function. When the button is clicked, the login process is initiated. The GIDSignIn.sharedInstance() accesses the

shared instance of the GIDSignIn class, which handles sign-in interactions with Google. Finally, the signIn.signIn() triggers the Google Sign-In process, presenting the user with the familiar Google sign-in screen.

The next stanza of code handles the Google sign-in results in the completion block. The signIn(withPresenting:) call invokes the completion handler when the sign-in process completes, either successfully or with an error. Finally, the error check prints errors to the console if any.

On a successful sign-in, `user.userID` retrieves the user's unique Google ID and `user.profile?.email` retrieves the user's email address and then displays both in print statements.

Passkey Support

Most login systems use a username and password. That won't always be the case. Apple, Google, and Microsoft worked together to embrace an existing protocol called WebAuthn, to help usher in what they called the "passwordless future."

Passkeys leverage cryptography and biometrics (Touch ID or Face ID) to eliminate reliance on passwords, a vulnerable target for phishing and hacking. This allows users to effortlessly sign in with a single tap or glance, minimizing friction and frustration compared to remembering and typing complex passwords. Passkeys seamlessly synchronize across Apple devices, ensuring consistent accessibility and convenience – and with a QR code, users can even share passkeys to other platforms. In Figure 4-5 you can see a passkey being set up on a web page.

Apple has extensive documentation for passkey integration available at https://developer.apple.com/documentation/authenticationservices/connecting_to_a_service_with_passkeys.

Note This section is for adding passkey support to an app, not acting as a passkey provider.

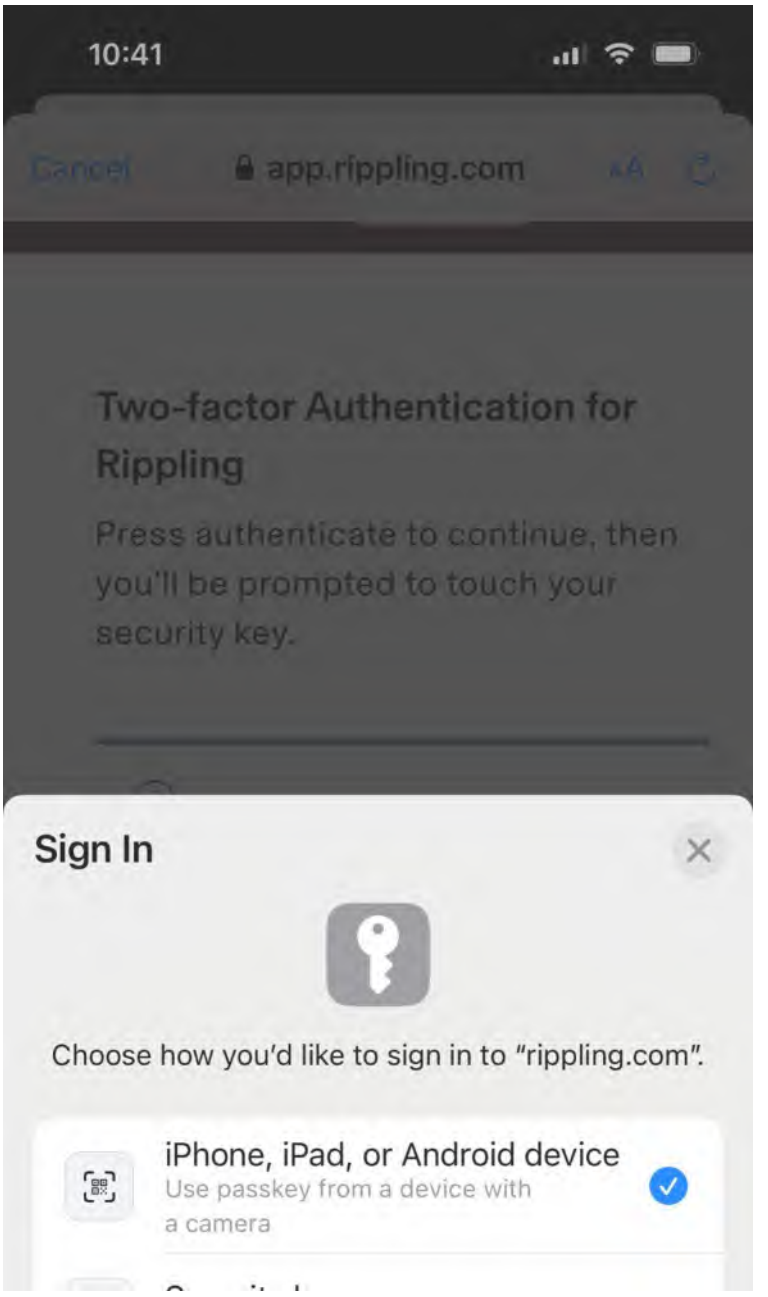

Figure 4-5. *Using Two-factor authentication*

Adding passkeys to a swift app involves first implementing Apple's Authentication Services framework, or more specifically ASAuthorizationPlatformPublicKeyCredentialProvider and ASAuthorizationController, used to manage passkey creation and authentication flows.

Then developers need to configure the sign-in options, which involves configuring supported credential types for sign-in, including passkeys alongside traditional usernames and passwords (which, e.g., are used to authenticate when the lid to laptop is closed).

Once compatibility for passkeys has been enabled, it's then possible to display your app's sign-in button and properly handle passkey creation. Here, respond to user selection of the sign-in button by initiating the passkey workflow. Guide users through the process of creating a passkey specific to your app, ensuring they understand the benefits and implications.

Authenticating with passkeys uses ASAuthorizationPlatformPublicKeyCredentialAssertionRequest to verify existing passkeys for users returning to an app. Touch ID or Face ID prompts for secure biometric verification and eliminates the need for password recollection, although a PIN can be used to bypass the biometric requirement. Once entered, it's then necessary to manage the passkey state. This involves implementing logic to handle scenarios like passkey deletion or device change.

Consider providing information and options to reestablish access on alternate devices.

Now that the flow is defined, let's put all this in action in an app, starting with enabling passkey support by importing the necessary framework, AuthenticationServices:

```
import AuthenticationServices
```

Next, create a sign-in request:

```
let provider =
ASAuthorizationPlatformPublicKeyCredentialProvider(relyingPartyIdentifier:
"example.com")
let request = provider.createCredentialAssertionRequest(challenge:
challenge)
```

Now that there's a sign-in request, present the request:

```
let controller =
ASAuthorizationController(authorizationRequests: [request])
controller.delegate =
self
```

```
controller.presentationContextProvider =
self
controller.performRequests()
```

Once the request has been interacted with, it's time to handle the authentication results from the passkey provider:

```
extension YourViewController: ASAuthorizationControllerDelegate
{
    func
authorizationController(controller: ASAuthorizationController,
didCompleteWithAuthorization
authorization: ASAuthorization) {
        switch authorization.credential {
        case
let credential as
ASAuthorizationPlatformPublicKeyCredentialAssertion:
            // Process the passkey credential
            let credentialID = credential.credentialID
            let userId = credential.userID
            // ... (Handle passkey creation or authentication)
        default:
            break
        }
    }
    func
authorizationController(controller: ASAuthorizationController,
didCompleteWithError
error: Error) {
        // Handle errors
        print("Authentication error: \(error)")
    }
}
```

Then conform that to the presentation context provider:

```
extension YourViewController:
ASAuthorizationControllerPresentationContextProviding
{
    func
presentationAnchor(for
controller: ASAuthorizationController) -> ASPresentationAnchor {
        // Provide a view for presenting the authentication UI
        return self.view.window!
    }
}
```

The challenge in the above code is a unique value provided by the server for passkey registration or authentication. The ASAuthorizationController manages the passkey authentication flow. The ASAuthorizationPlatformPublicKeyCredentialAssertion represents the passkey credential in this flow. The ASAuthorizationControllerDelegate handles authentication results and errors.

The ASAuthorizationControllerPresentationContextProviding provides the context for presenting authentication UI to users. On the server, it's necessary to handle passkey registration and authentication logic, beyond the scope of this chapter.

Note As with the other sections for authentication and authorization, each app is different and so developers will need to adapt it to your specific app's requirements and server-side configuration.

Testing and debugging passkeys, by intent, can be a little tricky. Thoroughly test passkey integration on various devices and iOS versions to ensure seamless functionality. Also check out multiple sites to make sure Google, Apple, and a couple of others work with each change. Different implementations can have different options. The authors of this book open sourced some of our Passkey research at https://github.com/krypted/webauthn-inspector. This is a Chrome extension that allows admins to see the raw data sent and received from sites that support passkeys. Finally, if passkeys are used on a site, make sure to maintain secure server-side handling of passkey data and adhere to Apple's best practices for authentication.

Sometimes it feels like developers are flying the passkey plane while it's being built. This is fairly new technology and there are lots of changes being made as this is being written. So keep abreast of Apple's passkey documentation at https://developer.apple.com/documentation/authenticationservices/public-private_key_authentication/supporting_passkeys/ and their authentication services framework at https://developer.apple.com/documentation/authenticationservices and check https://developer.apple.com/documentation/authenticationservices/connecting_to_a_service_with_passkeys for updated sample code.

Web Gateways

This chapter has primarily discussed keys. Sometimes, those keys get used in apps. However, the authentication against a web gateway might first be a username and a password, which then returns a key of some kind, often decoded as a JSON object. Let's look at how this specific process might work:

```
import Foundation
func authenticateAgainstWebGateway(username: String, password: String,
completion: @escaping (Bool) -> Void) {
  let url = URL(string: "https://example.com/api/auth")!
  var request = URLRequest(url: url)
  request.httpMethod = "POST"
  request.setValue("application/json", forHTTPHeaderField: "Content-Type")
  let body = ["username": username, "password": password]
  request.httpBody = try? JSONEncoder().encode(body)
  let task = URLSession.shared.dataTask(with: request) { data, response,
  error in
    if let error = error {
      print(error)
      completion(false)
      return
    }
    if let response = response as? HTTPURLResponse, response.status
    Code != 200 {
      print("Error: \(response.statusCode)")
```

```
        completion(false)
        return
      }
      if let data = data,
        let json = try? JSONDecoder().decode(AuthResponse.self, from: data) {
        if json.success {
          completion(true)
        } else {
          print(json.error ?? "Authentication error")
          completion(false)
        }
        return
      }
      completion(false)
    }
    task.resume()
}
struct AuthResponse: Codable {
    let success: Bool
    let error: String?
}
```

This code creates a function that takes a username and password as input and calls a completion handler with a boolean value indicating whether or not the user was successfully authenticated against the web gateway. The function first creates a URLRequest object and sets the httpMethod property to POST. The Content-Type header is set to application/json to indicate that the body of the request is JSON data.

The body of the request is then encoded as JSON and set as the httpBody property of the URLRequest object. A URLSession task is then created and the resume() method is called. The task closure will be executed when the request completes.

The task closure checks the response status code and calls completion(false) if the status code is not 200. If the response data is not nil, the closure decodes the data as an AuthResponse object and calls the completion handler with the success property of the object.

Federation

OpenID Connect (OIDC) and Security Assertion Markup Language (SAML) are two different authentication protocols that can be used to allow users to sign in to websites and applications. OIDC is a newer protocol that is based on OAuth 2.0, while SAML is an older protocol that is based on XML.

OIDC is a lightweight protocol that is designed to be easy to implement. It uses JSON web tokens (JWTs) to exchange authentication information between the user's browser and the website or application. OIDC is often used in conjunction with OAuth 2.0 to allow users to grant access to their accounts to third-party applications.

SAML is a more complex protocol that uses XML to exchange authentication information between the user's browser and the website or application. SAML is often used in enterprise environments where security is a top priority.

The best protocol for each environment will depend on the specific needs. When looking for a lightweight protocol that is easy to implement and modern, OIDC is probably a good choice - especially if SAML isn't required from a Federated Identity Provider (IdP). Because all of this can require a lot of knowledge, Okta paid stupid amounts of money to buy a company that made it easy: Auth0.

Using Auth0 to provide OIDC and SAML federation

Auth0 serves as a centralized identity provider, managing logins and user profiles for your applications. It offers both native and social login options, providing users with a convenient and consistent experience.

OIDC: Streamlined Authentication for Modern Apps

OIDC offers a lightweight and standardized approach to authentication, using JSON Web Tokens (JWTs) to securely share user information between your application and Auth0.

Integrating OIDC with Swift is straightforward, utilizing Auth0's Swift SDK. With just a few lines of code, you can handle login, validate tokens, and retrieve user data.

Here's a glimpse of how OIDC integration works in Swift:

```
import Auth0

Auth0
    .webAuth()
    .scope("openid profile email")
```

```
    .start { result in
        switch result {
        case .success(let credentials):
            // Use obtained credentials
            _ = credentials
        case .failure(let error):
            // Handle error
            print(error)
        }
    }
```

SAML provides a robust, XML-based standard for federation as well, also allowing apps to seamlessly integrate with existing identity providers (IdPs). Auth0 acts as a SAML service provider (SP), facilitating the communication between the app and the IdP. Setting up SAML federation with Auth0 can be fairly straightforward, as their SDK provides helpful tools for building SAML requests and handling responses.

Here's a simplified example of initiating a SAML login once the Auth0 API is imported:

```
import Auth0

Auth0
    .webAuth()
    .connection("YOUR_SAML_CONNECTION_NAME")
    .scope("openid profile email")
    .start { result in
        // Handle response and retrieve user information
    }
```

Keep in mind that OIDC and SAML are usually about the same amount of difficulty to get working in an app, especially when using a framework like Auth0 to get a bunch of the necessary bits for "free" (which is obviously relative). If working at a company that has OIDC or OAuth rolled out elsewhere, maybe consider that first. If the company has a lot of SAML infrastructure, consider that. If neither, it's probably best to start with OIDC as a consideration, as it's ideal for modern applications built for user convenience and integration with social logins. Its lightweight nature makes it a good choice for mobile apps and API access.

This chapter hasn't gotten into user management, as this just scope creeps the thing too far. Given that the book is for admins, it's likely that there will be others out there who handle user management, especially in bigger customers. That involves things like role-based access, attributes for accounts, enforcing security policies, a rules engine that includes onboarding, multifactor authentication, and integration with other backend systems. If the company doesn't yet have an IdP, then there are lots of questions there that are going to need to be asked.

Summary checkpoint: At this stage, the chapter has moved from token mechanics into identity-provider workflows (social login, passkeys, OIDC, and SAML) so readers can compare implementation options by environment and user needs.

Remember This is a general overview. Refer to Auth0's official documentation for detailed instructions and code examples for your specific setup.

Cookies

Cookies are small bits of data that websites store on devices when users visit them. These files can contain information about browsing activity on that specific website. Each cookie typically contains a key-value pair, where the key identifies the cookie and the value stores the information.

Security Note: Avoid storing sensitive data in cookies unless it is strictly necessary and properly protected with `Secure` and `HttpOnly` flags (and appropriate `SameSite` settings).

A website developer can define cookies and what information they should store. When users visit a website that uses cookies, it sends a cookie request to the device. The web browser receives the cookie request and, if allowed, stores the cookie on the device. "If allowed" because users can disable cookies globally or for a given website domain. When the user visits the same website again, the browser automatically sends the stored cookies back to the website. The website can then use the information in the cookies to remember any preferences that were configured, personalize the experience the user gets on the site, and track activity on the site.

These keep users logged into sites. Cookies can store login credentials so users don't have to enter them every time they visit a website. Cookies can store your language preferences, theme choices, and other settings on a website. Cookies can also be used

to remember the items users have added to a shopping cart on an e-commerce site. All of this sounds useful, and all of this can still work with sites that use more modern authentication techniques, like OAuth. However, cookies can be used to track browsing history on a website, which can be used for targeted advertising or analytics. This is where privacy concerns can come in. However, not all cookies are the same. Here's a rundown of the different types of cookies:

- *Session cookies*: These cookies are temporary and are deleted when a web browser is closed.
- *Persistent cookies*: These cookies stay on devices for a longer period of time, even after a browser is closed.
- *First-party cookies*: These cookies are created by the website a user is visiting.
- *Third-party cookies*: These cookies are created by other websites or domains embedded on the website a user is visiting.

Users can then control how cookies are used on their devices through the settings for each web browser. All cookies can be blocked or accepted, or managed on a site-by-site basis. To see how easy it is to enable or disable cookies in Safari, simply open Safari, go to Settings, and then click the Advanced tab. Here, as seen in Figure 4-6, users can simply "Block all cookies," and many do.

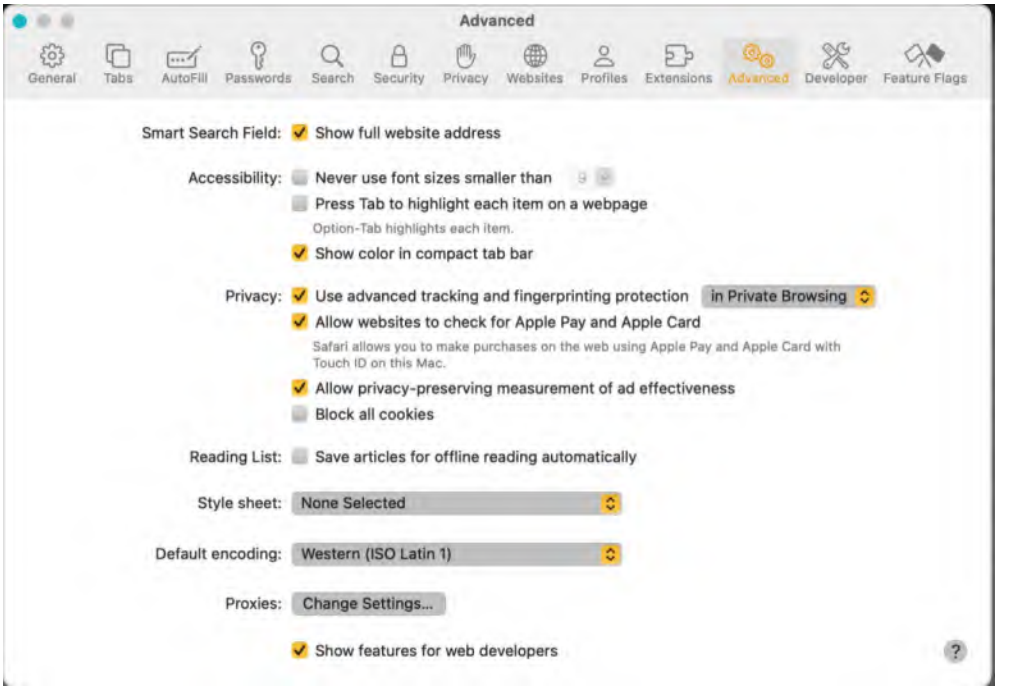

Figure 4-6. *Safari Advanced settings*

To see if a site has cookies on a computer in Safari, open Safari, go to Settings, and click Privacy. Then click Privacy and search for a site. As seen in Figure 4-7, notice that Apple uses cookies.

Figure 4-7. *Editing website data in Safari*

Ironically, Apple has long complained about the rampant use of cookies to violate the privacy of users. The rhetoric is mostly true – cookies can be used for purposes other than why users often approve them to be created. For a while, it felt like Apple was the only one thinking about how cookies impacted user privacy; however, that has all been changing in recent months. As of the time of this writing, Chrome has been rolling out restrictions on third-party cookies.

This is all covered because plenty of apps might use a webkit view to interact with cookies. Swift also relies on the shared HTTPCookieStorage class to manage cookies. This singleton handles all cookie reading, writing, and deletion for apps. To fetch cookies for a specific URL, use the cookies(for:) method on the HTTPCookieStorage object, passing the target URL as an argument. This returns an array of HTTPCookie objects, each containing the cookie's name, value, domain, and other properties:

```
let url = URL(string: "https://www.example.com")!
let cookies = HTTPCookieStorage.shared.cookies(for: url) ?? []
for cookie in cookies {
  print("Name: \(cookie.name), Value: \(cookie.value)")
}
```

Create a new HTTPCookie object with the desired properties like name, value, domain, and path to send a new custom cookie request. Then, use HTTPCookie.requestHeaderFields(with:) to add the appropriate Cookie header to a URLRequest, as follows:

```
let cookieName = "mySuperCookie"
let cookieValue = "secretDeliciousness"
let cookie = HTTPCookie(properties: [
  .domain: url.host!,
  .path: "/",
  .name: cookieName,
  .value: cookieValue
])!
var request = URLRequest(url: url)
request.allHTTPHeaderFields = HTTPCookie.requestHeaderFields(with:
[cookie])
// Send the request with your custom cookie!
```

To allow an app to use cookies in web content, use WKWebView, which has its own cookie store via WKWebsiteDataStore. Read/write cookies through WKHTTPCookieStore. However, remember that cookies can be intrusive and track user data – so Apple continues to refine the rules around their access. Always treat them responsibly to future-proof getting in trouble with Apple. Obtain user consent for cookie usage, ensure proper security measures, and follow relevant privacy regulations.

Because cookies can be used but aren't recommended, this book isn't going to go into some of the more advanced concepts. Managing cookie storage, handling third-party cookies, and integrating with custom cookie libraries might be required in environments where they're necessary. This was covered because it's a good idea to understand what these are and how they're used, but read up at HTTPCookieStorage and WKWebView for more advanced framework uses.

Conclusion

There are about as many ways to authenticate and authorize users as there are programmers that truly understand what's happening behind the scenes. The landscape of how to get people logged into sites and services is also changing about as fast as it's ever changed in the history of computing. This means that whatever is done in apps today is likely to be technical debt within a year or three. Therefore, do whatever can be done to make apps secure and stable; however, also consider that the time investment and distribution of tools (and time to distribute tools) will need to be commensurate over time as well.

CHAPTER 5

Notifications and Data Flow

Notifications and data flow are the lifeblood of any dynamic app. Managing notifications means that it's possible to communicate with a device when the app isn't open. Once the app is opened, it should respond smoothly to user actions, update views with fresh data, and keep users informed and engaged.

Notifications come in two flavors: local and remote. Local notifications are triggered within an app, like task reminders or download completions. Remote notifications, delivered through Apple Push Notification service (APNs), update users even when the app is inactive, so it can be activated to perform a task. Both require proper handling to deliver relevant information without bombarding users.

This chapter is organized into four practical tracks so the scope stays clear:

1. Notifications and APNs fundamentals
2. Debugging notification delivery
3. Deep linking into app views
4. Backend/serverless data-flow patterns that support notifications

Data flow refers to the path information takes through an app – from fetching to manipulation to presentation in the UI. Maintaining a clear and efficient flow is crucial for performance and stability. Here are some key tools:

- *Property wrappers*: SwiftUI provides property wrappers like @State and @ObservedObject for managing data flow within views. @State stores temporary data changes within a single view, while @ObservedObject manages shared data across multiple views.

C. Edge et al., *Beyond Scripting*, https://doi.org/10.1007/979-8-8688-3035-8_5

- *Observable objects*: These conform to the ObservableObject protocol, and their published properties automatically trigger view updates when they change. This keeps the UI in sync with underlying data seamlessly.
- *MVVM and MVC architectures*: Model-View-ViewModel (MVVM) and Model-View-Controller (MVC) architectures separate data handling, UI logic, and user interaction, promoting cleaner code and easier data flow management.
- *Data stores*: Implement dedicated data stores like Core Data or local databases to persist and manage complex data structures throughout your app.

There are a few techniques to optimize data flow. Use libraries like Combine or SwiftNIO to handle asynchronous tasks and data streaming without writing cumbersome code. This reduces boilerplate code and reinventing the wheel. Don't overwhelm users with excessive notifications. Use time intervals or context-aware triggers to send notifications at opportune moments. Design notifications that are relevant, timely, and actionable. Provide clear information and intuitive ways for users to interact with them. Experiment with different notification formats and content to find what resonates with users.

Explore tools like SwiftUI bindings and environment objects for dynamic data sharing between views. But more importantly, look at some of the awesome work done by other developers, like the swiftDialog project, available at https://github.com/swiftDialog/swiftDialog (Figure 5-1).

Figure 5-1. *swiftDialog*

If a project like swiftDialog does what's needed, don't write a custom tool. But going beyond simply displaying dialogs and providing push notifications to bring an app into the foreground, it helps to understand what exactly a push notification is.

APNs

Push notifications are messages that are sent to Apple devices, even when the app is not running. They are a great way to keep users engaged with an app and to let them know about important updates (like running their own software updates). Push notifications work by using a technology called Apple Push Notification service (APNs). APNs is a secure, reliable, and efficient way to send messages to Apple devices. That security can be kinda' annoying when setting up a new service, but it's why it's so great.

When a user registers for push notifications, their device generates a unique identifier called a device token. This token is used to identify the device and the application to send push notifications to it. When an app service sends a push notification, a message is sent to the APNs. The APNs then delivers the message to the device with the matching device token. The device wakes up and displays the push notification to the user.

APNs delivery flow at a glance:

1. App requests notification permission from the user.
2. Device receives a device token for that app.
3. App sends the device token to your backend/provider server.
4. Backend sends a payload to APNs.
5. APNs delivers the notification to the target device.

To enable push notifications for an app, the following will be necessary:

- Register your app with APNs.
- Add the required framework (such as UserNotifications) to the app.
- Implement the UNUserNotificationCenterDelegate protocol.
- Request permission from the user to receive push notifications.
- Send push notifications to the user.

Before registering with APNs and trying to send a message, it's important to understand how all this stuff actually works.

APNs in an App

Device tokens and topics are two key concepts in Apple Push Notification service (APNs). A device token identifies a specific app-device pair, and a topic identifies the destination app.

There are a few elements of an APNs message. A device token identifies where the message is sent, and the `apns-topic` header identifies which app should receive it. APNs itself does not maintain end-user subscriptions to arbitrary topics like "news"; if an app needs that behavior, the app's backend maintains it.

Each app needs to be registered with APNs. To do so, enable Push Notifications in the app's signing/capability settings. On the provider side, APNs authentication is done with either a token key (`.p8`) or a provider certificate (`.p12`). We'll dig into how to do this later in the chapter.

The next step is to implement the UNUserNotificationCenterDelegate protocol. This protocol lets your app handle notification presentation/interaction, and you should also register for remote notifications to receive an APNs device token. To implement this, you can add code like the following to your app delegate:

Runs in the app:

```
import UIKit
import UserNotifications
class AppDelegate: UIResponder, UIApplicationDelegate,
UNUserNotificationCenterDelegate {
  func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [UIApplication.
LaunchOptionsKey: Any]?) -> Bool {
    // ...
    // Register for push notifications
    UNUserNotificationCenter.current().delegate = self
    UNUserNotificationCenter.current().requestAuthorization(options:
    [.alert, .sound, .badge]) { granted, error in
      if granted {
        // Push notifications are enabled
        DispatchQueue.main.async {
          application.registerForRemoteNotifications()
        }
      } else {
        // Push notifications are not enabled
      }
    }
    return true
  }

  func application(
    _ application: UIApplication,
    didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data
  ) {
    // Send `deviceToken` to your provider server
  }
  // ...
}
```

The next step is to request permission from the user to receive push notifications. That involves calling the requestAuthorization() method on the UNUserNotificationCenter object. The final step is to send push notifications to the user from a provider server by sending a POST request (HTTP/2) to APNs.

Here is an example of how to send a push notification:

Runs on a provider server/backend:

```
func sendRemoteNotification(deviceToken: String, providerToken: String)
async throws {
  let url = URL(string: "https://api.push.apple.com/3/
  device/\(deviceToken)")!
  var request = URLRequest(url: url)
  request.httpMethod = "POST"
  request.setValue("bearer \(providerToken)", forHTTPHeaderField:
  "authorization")
  request.setValue("com.example.myapp", forHTTPHeaderField: "apns-topic")
  request.setValue("alert", forHTTPHeaderField: "apns-push-type")
  request.setValue("application/json", forHTTPHeaderField: "content-type")

  let payload: [String: Any] = [
    "aps": [
      "alert": [
        "title": "My Notification",
        "body": "This is my notification"
      ]
    ]
  ]
  request.httpBody = try JSONSerialization.data(withJSONObject: payload)

  let (_, response) = try await URLSession.shared.data(for: request)
  guard let http = response as? HTTPURLResponse, http.statusCode ==
  200 else {
    throw NSError(domain: "APNs", code: 1)
  }
}
```

This code will send a push notification with the title “My Notification” and the body “This is my notification” to the specified device token. This should run on a provider server (not inside the iOS app). There are a number of ways to send those push notifications. One of the easiest is using Amazon’s Simple Notification Service (SNS).

Security warning: Push payloads should contain references, IDs, or commands, not secrets. Do not place passwords, private keys, or other sensitive data directly in notification payloads.

Using AWS

AWS, or Amazon Web Services, is a cloud platform that offers a broad set of global compute, storage, database, analytics, application, and deployment services that help organizations move faster, lower IT costs, and scale applications.

To get started with SNS, AWS' Simple Notification Service which supports APNs, first create an AWS account. Go to the AWS website and click the “Create an AWS Account” button. Provide some basic information, such as name, email address, and billing information. Then start exploring the different AWS services that are available. Do so by browsing the AWS website or by using the AWS Management Console. There are plenty of different AWS services to choose from, and just giving these a quick review might save plenty of time in other parts of apps or services on a given roadmap.

One of the services is the Simple Notification Service, or SNS for short. Amazon SNS makes implementing Apple Push Notifications (APNs) a breeze. This might seem like a longer process when reading this section of the book, but it’s really not as many steps as it seems (although buttons on web pages move around a lot so look for specific words in button names). There’s a few main steps to go through: creating a cert in Keychain, generating a Push Notifications cert with the appropriate bundle ID and team ID, and adding an application instance. Notice that these are different for Mac and iOS, so if doing both, use iOS and if doing one for each, use the appropriate entry.

Create a Cert in Keychain

First, create a local certificate in the Keychain Access application as shown in Figure 5-2. This is used to generate an aps.cer on the Apple certificates portal (and is basically a throwaway once we’re done with the process). To start, open Keychain Access and under the Keychain Access menu, and select Certificate Assistant and then “Request a Certificate From a Certificate Authority…”.

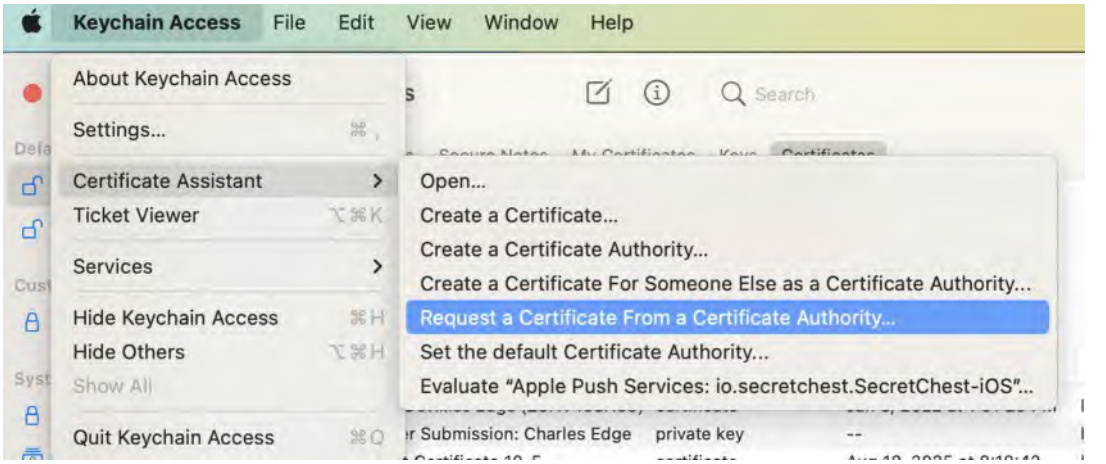

Figure 5-2. *Using Keychain Access to request a cert*

At the Certificate Assistant, Figure 5-3, choose Saved to disk" in the "Request is:" field. Provide a business email address and then provide a Common Name (e.g., App Name followed by Push Certificate). Click Continue.

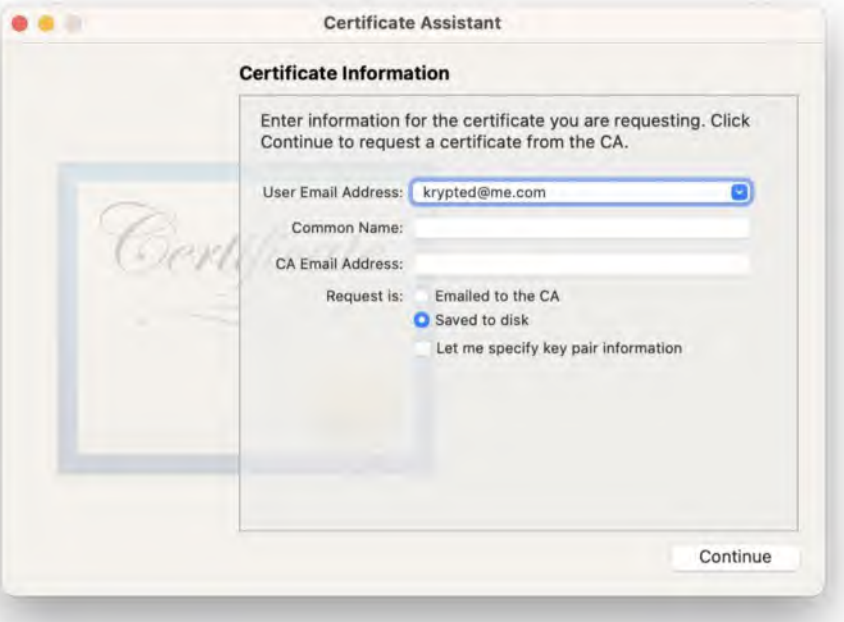

Figure 5-3. *Creating a cert using Certificate Assistant in Keychain Access*

When prompted for a place to save the certificate, Figure 5-4, choose a place that's easy to select in the next few steps and then click Save.

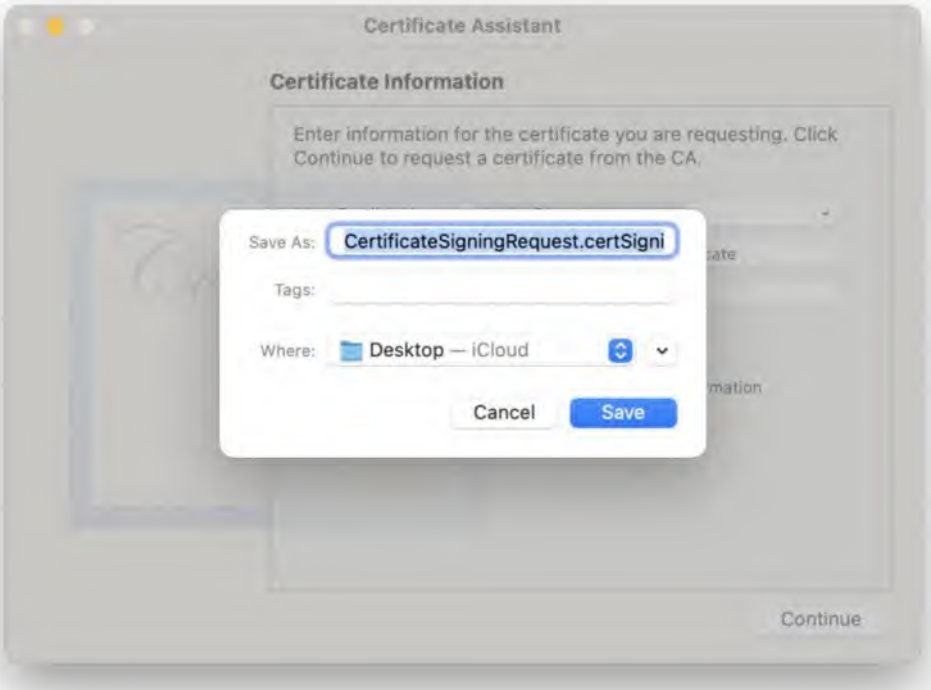

Figure 5-4. *Saving the certificate request*

Keep the certificate in a safe place, and don't give it to anyone. Once that step is done, it's time to go to the Apple Developer Portal to create an actual push certificate.

Create a Push Certificate for the App

To create a push certificate for an app, log into the Apple Developer Portal (developer.apple.com). Scroll to the footer of the page and click "Certificates, IDs, & Profiles" as seen in Figure 5-5.

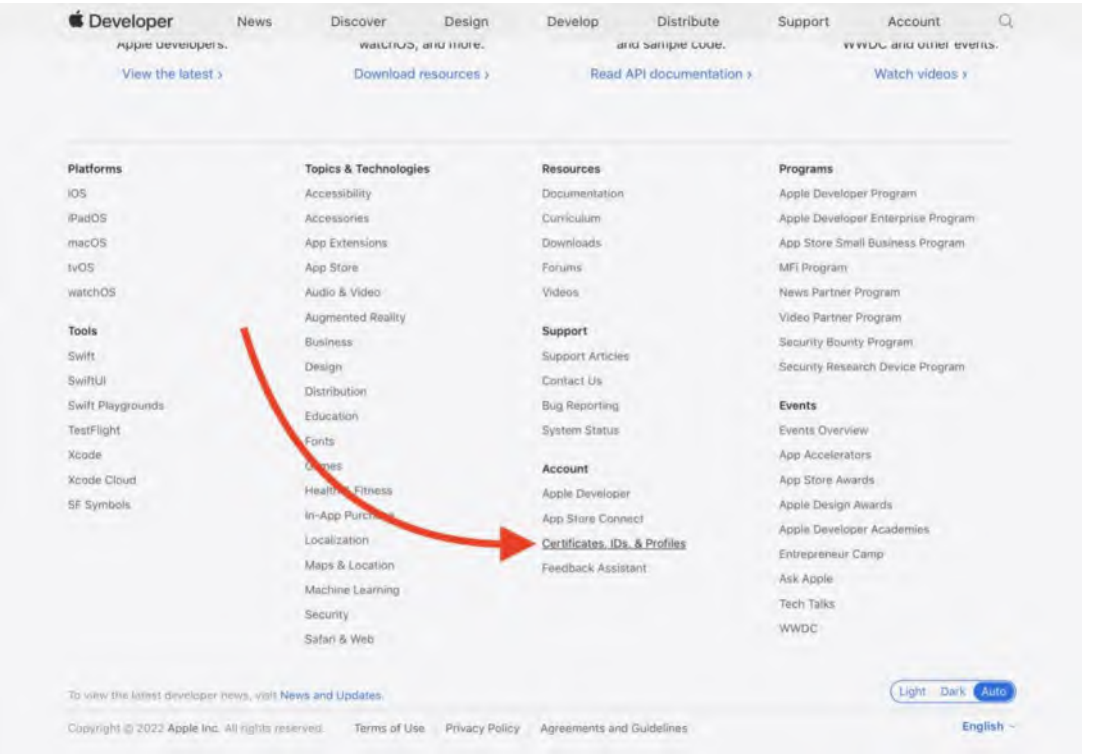

Figure 5-5. *Certificates, IDs & Profiles*

At the Certificates, Identifiers, & Profiles Screen, click Certificates, and then click on the plus sign to create a new cert. There are a lot of types of certificates, and for some developers, there are a lot of apps so make sure these next steps match up properly. First, scroll down to the "Apple Push Notification service SSL (Sandbox & Production)" option in Services (Figure 5-6). This specifies that the cert will be used to send an APN (or hundreds of thousands of them). Click Continue.

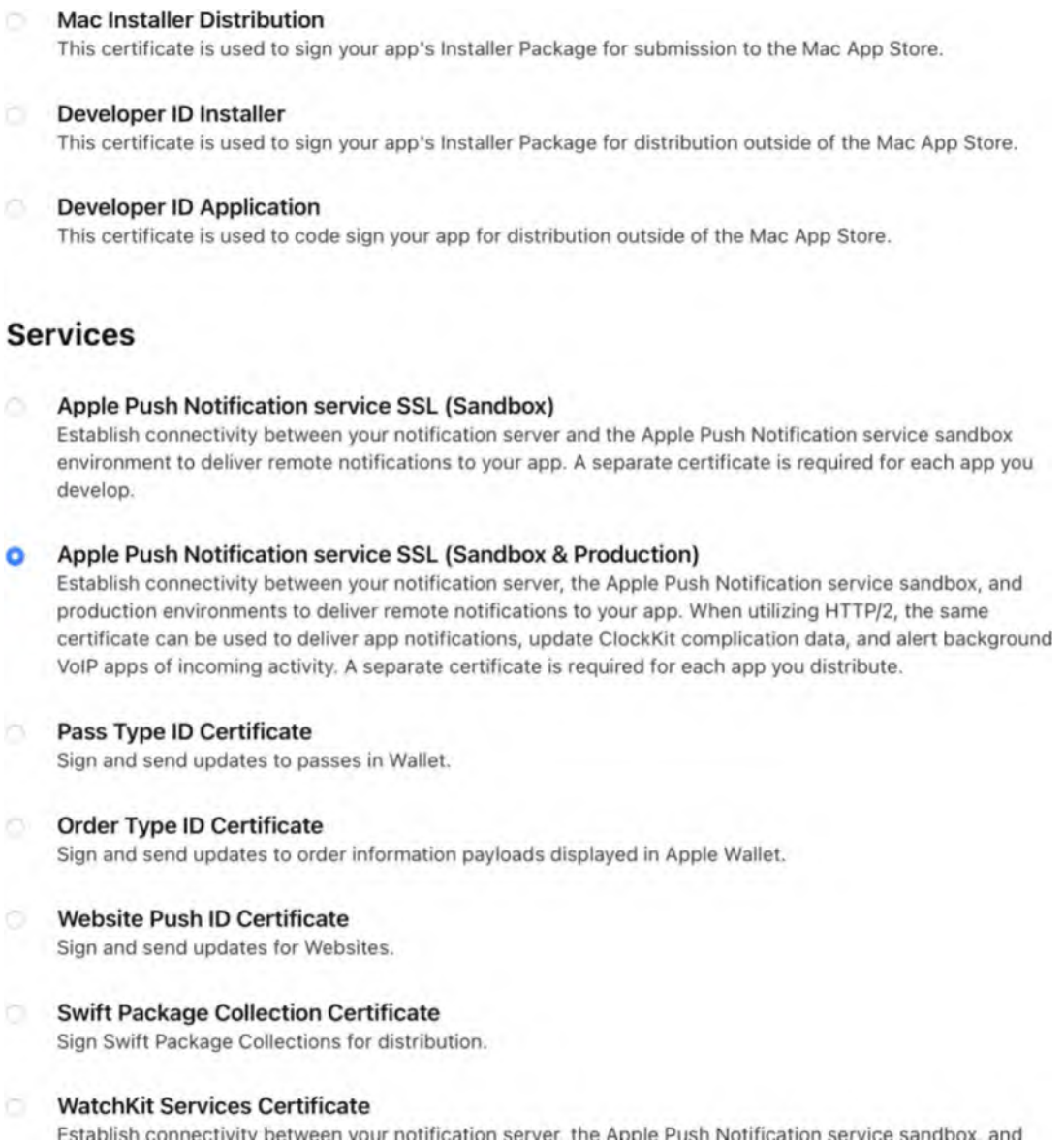

Figure 5-6. *Creating a certificate for APNS*

Click Choose File (Figure 5-7), and when prompted, select the certificate created in the previous section, which if formatted properly will appear in the dialog once selected. Click Continue.

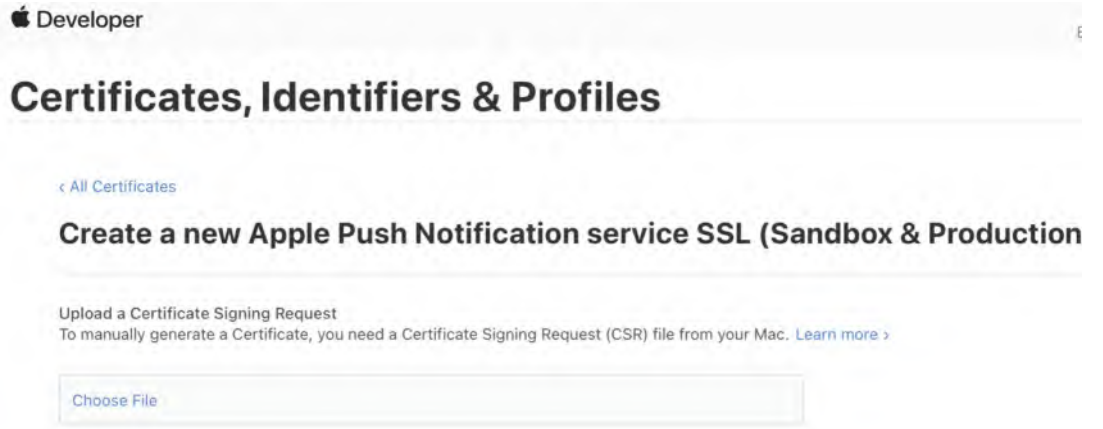

Figure 5-7. *Upload a certificate signing request*

Once the certificate is created, click Download to download a .cer to the computer as seen in Figure 5-8. Keep this file safe.

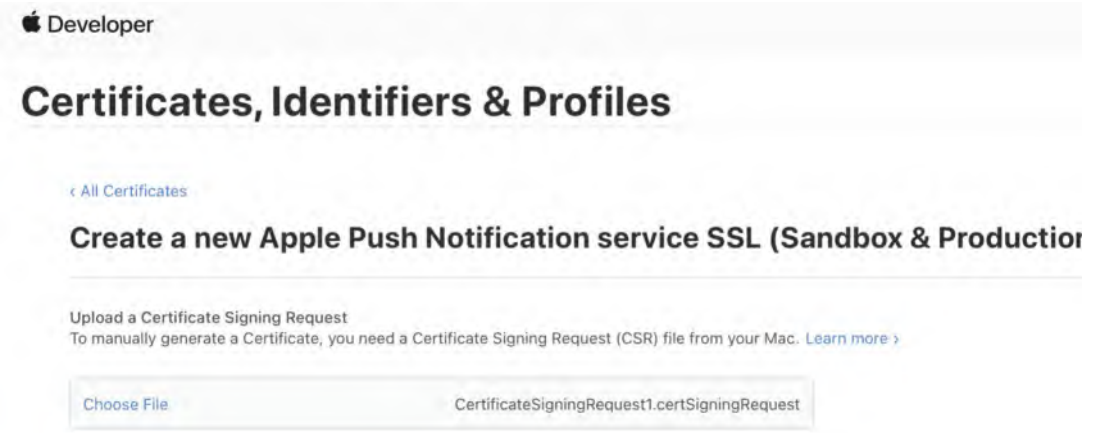

Figure 5-8. *Download the generated certificate*

Once the certificate is downloaded, it will be a .cer but we need a .p12 to load into AWS (or a .p8 if doing their token-based flow). Double-click the cert to import it into Keychain Access, and then control-click (or right-click if that's an option) on the cert, and click Export for the certificate as seen in Figure 5-9. Alternatively, certs can be converted to a .pfx/certificate pair with the openssl command, but this way is easier.

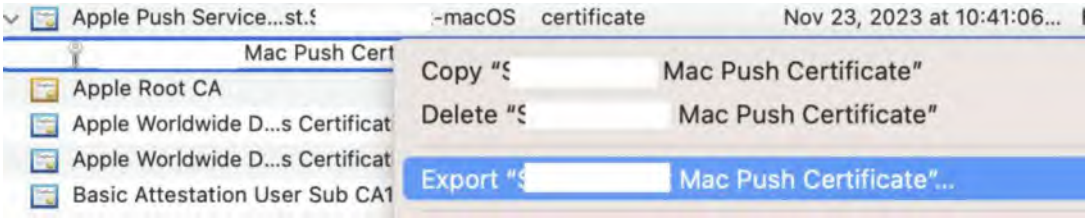

Figure 5-9. *Exporting a certificate from Keychain Access*

Select a place to export the certificate to, leaving the File Format set to .p12 as seen in Figure 5-10, and click Save.

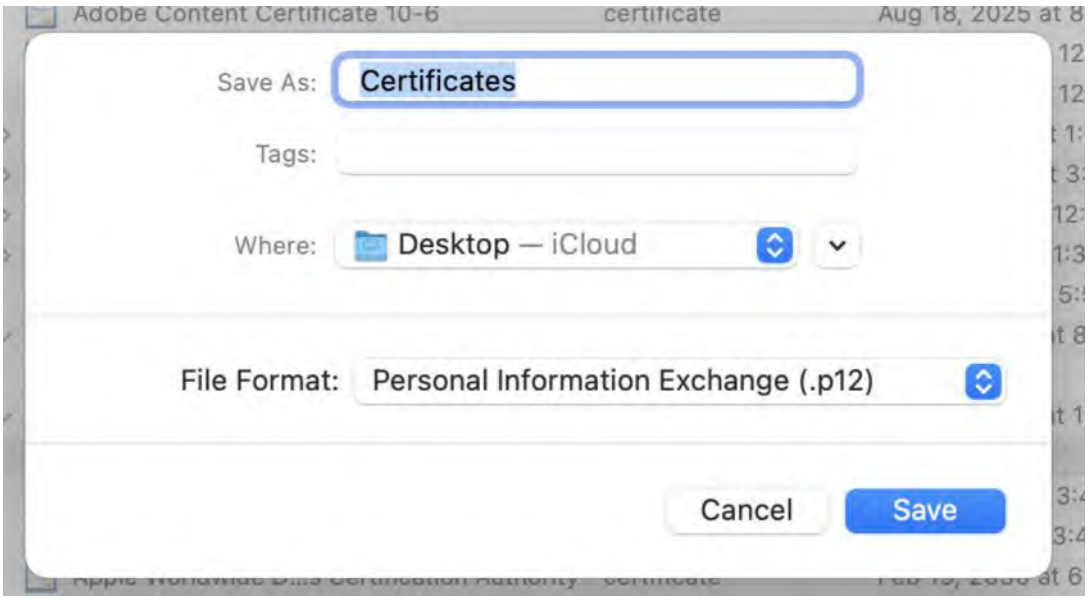

Figure 5-10. *Choosing a location to export the certificate*

Provide a password for the certificate (this will be used to unencrypt the .p12 on the AWS site) as seen in Figure 5-11.

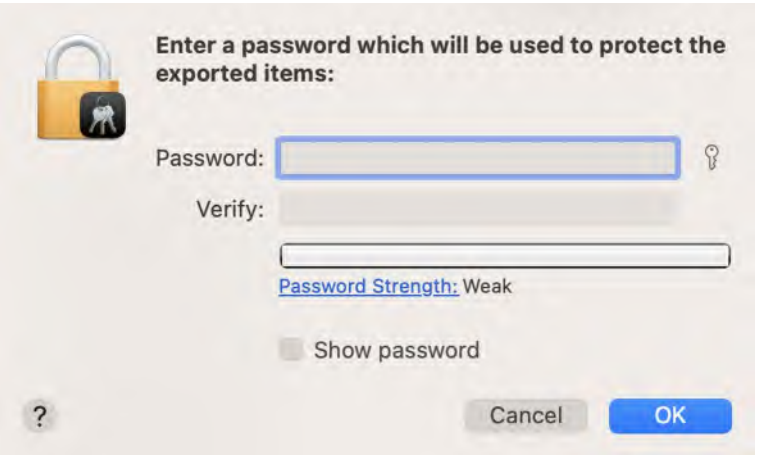

Figure 5-11. *Setting a password on the exported certificate*

Now it's time to head over to the AWS console at `http://aws.amazon.com` to set up an SNS instance.

Set Up AWS SNS

Now that there are certificates, it's time to configure SNS. At the AWS Console, select the Simple Notification Service (or SNS for short) with an account that has access to set up a new service (one that potentially has a billing consequence). At the Amazon SNS screen, click Push Notifications and then "Create platform application" to set up a new instance of SNS, as seen in Figure 5-12 (keep in mind that if there are multiple versions of the app like for iOS and Mac, then new certs will need to be generated and this whole article repeated for each).

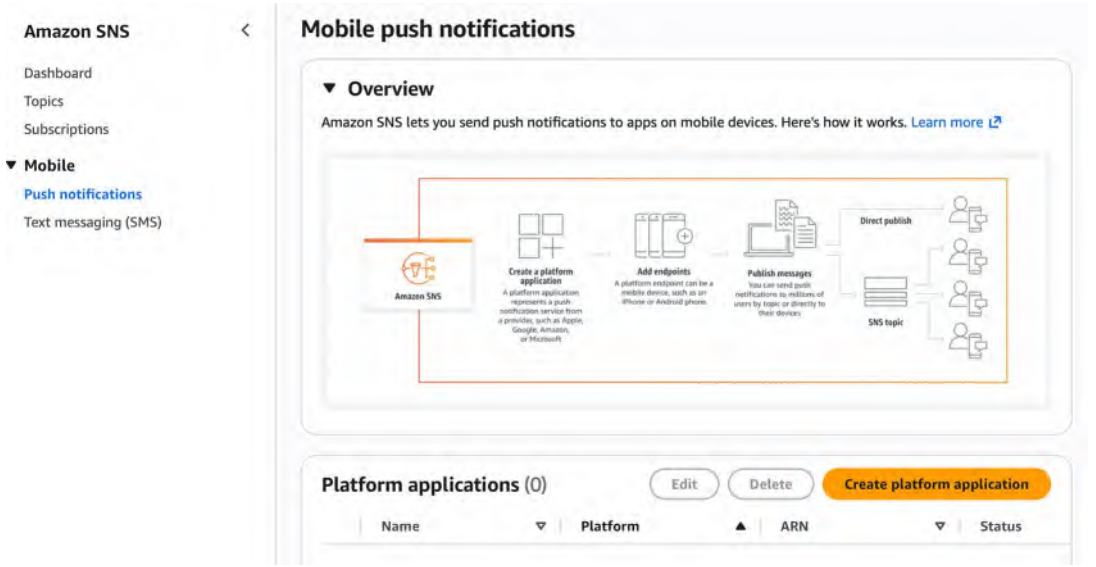

Figure 5-12. *Creating a new platform application in AWS SNS*

At the "Create platform application" screen, provide a name in the "Application name" (might wanna add iOS or Mac if there's one of each) and then select "Apple iOS/VoIP/MacOS" in the "Push notification platform" field (Figure 5-13).

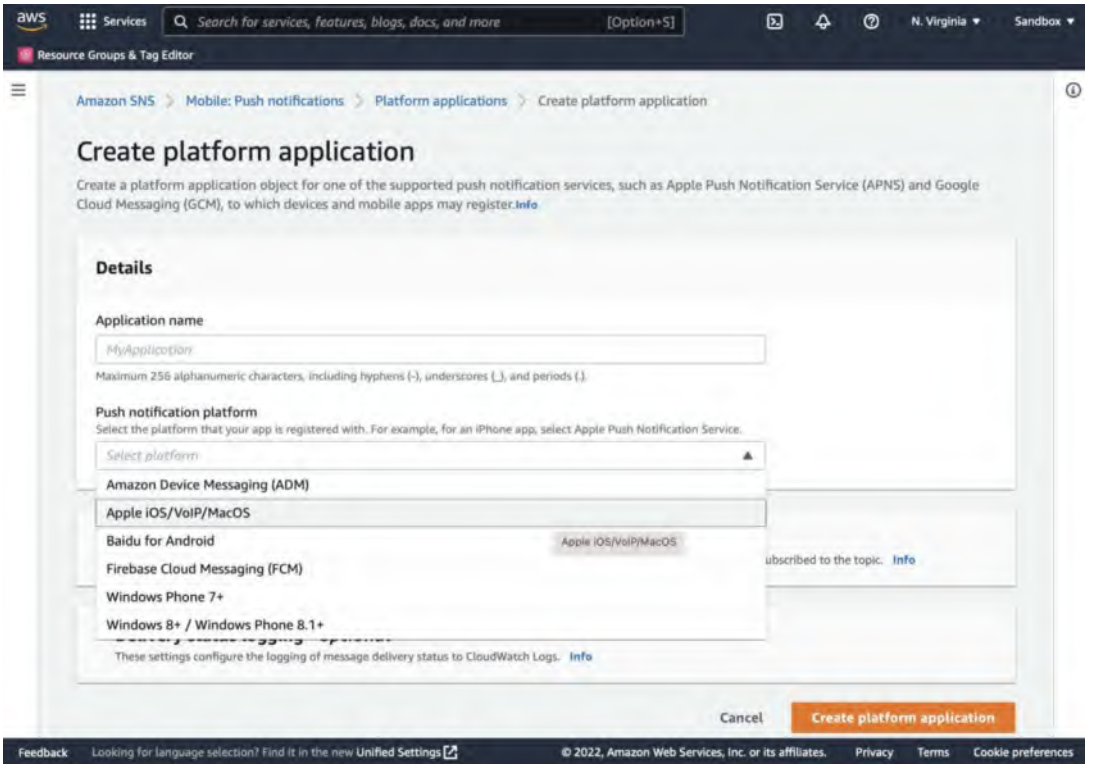

Figure 5-13. *Selecting Apple services in AWS SNS*

Select iOS or macOS in the "Push service" field, and select Certificate (there's a whole flow for Token-based stateless authentication but not really going there for this app). Click the Choose file box (Figure 5-14).

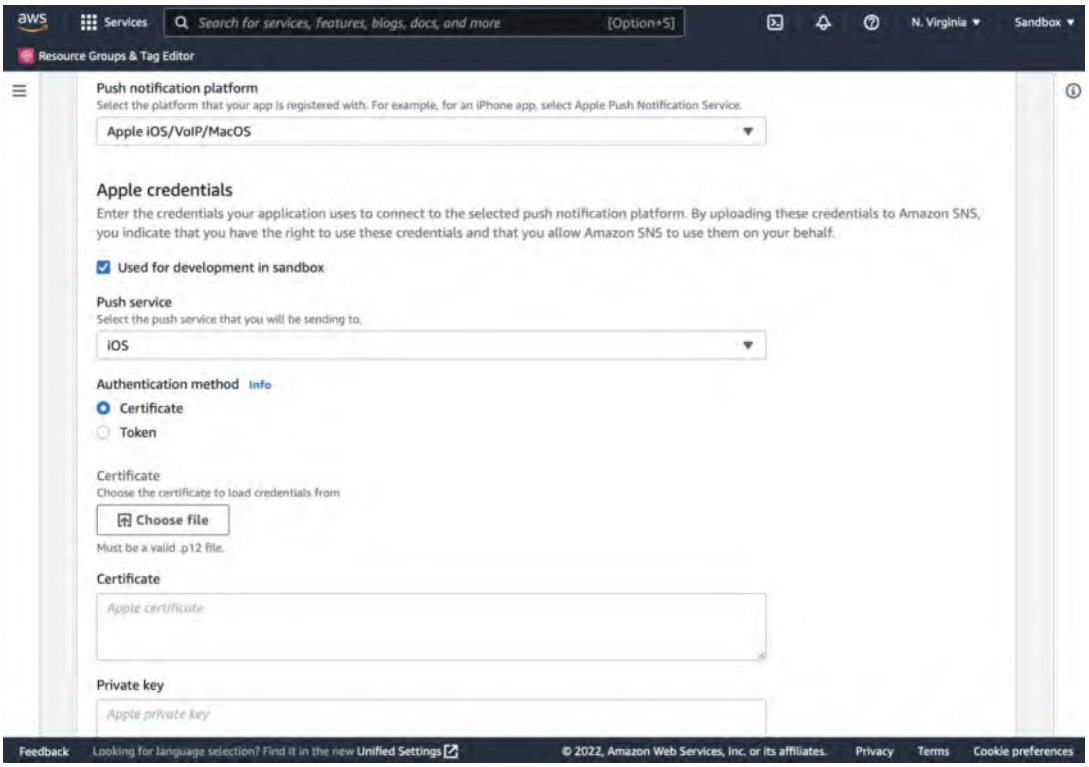

Figure 5-14. *Configuring Apple push notifications in AWS SNS*

At the browse dialog, select the .p12 cert created earlier and then click the Upload button (Figure 5-15).

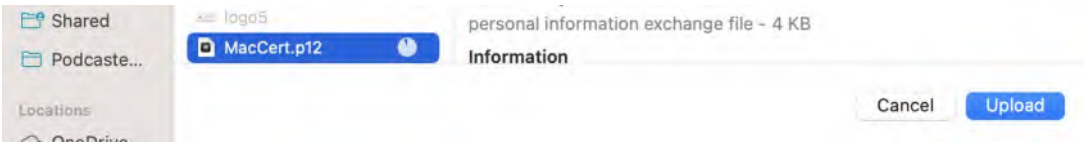

Figure 5-15. *Selecting the certificate*

Once there's a green checkbox by the name, provide the certificate password in the password field and then (and this is important) click the "Load credentials from file" button (Figure 5-16).

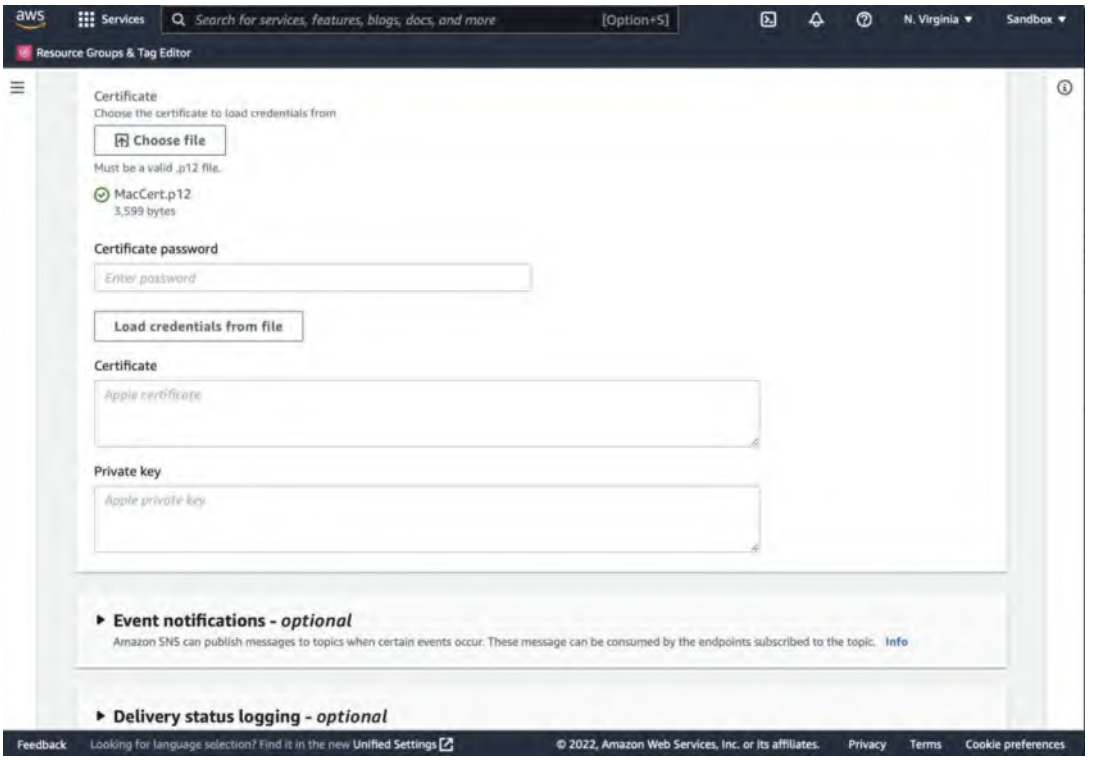

Figure 5-16. *Uploading credentials*

If the credential worked, the Certificate and "Private key" fields will each have the traditional BEGIN CERTIFICATE section followed by a big long string (Figure 5-17).

Figure 5-17. *Certificate fields*

The ARN is then listed in the dialog (this one is hidden but it's easy to copy and paste) along with the expiration date pulled from the cert that was imported (Figure 5-18). Admins can always use the Edit button to set up rates or CloudWatch integration for troubleshooting.

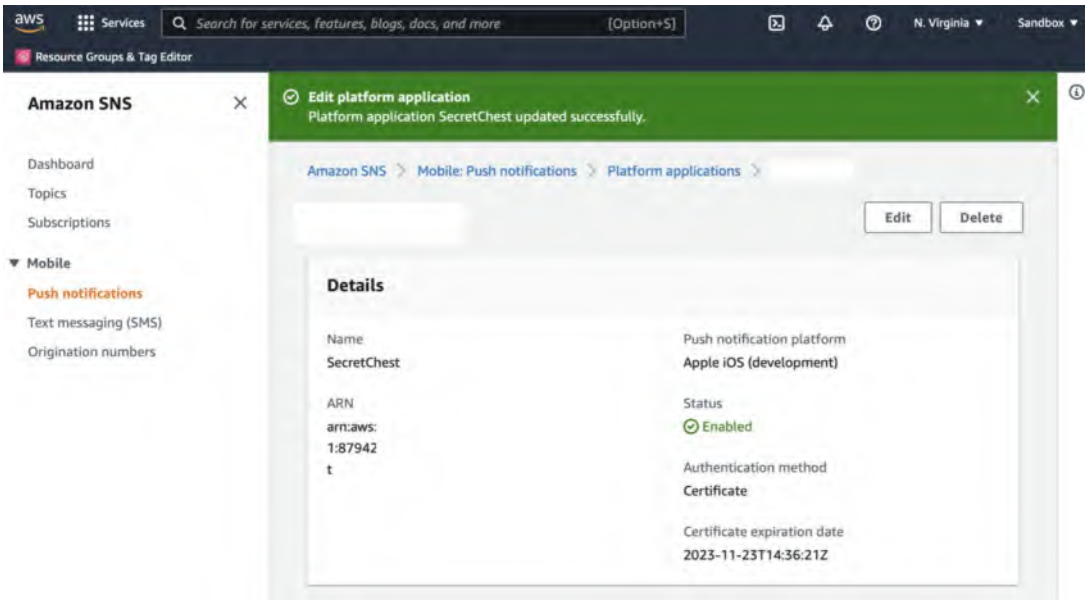

Figure 5-18. *Certificate enabled in AWS SNS*

Now that there's a cert and an ARN, check out Amazon's GitHub to set up Swift Package Manager to point to `https://github.com/aws-amplify/aws-sdk-ios-spm` per `https://github.com/aws-amplify/aws-sdk-ios` or use the AWS Amplify app to bring in the SDK. Import AWSSNS into the AppDelegate.swift:

```
import AWSSNS
```

Then there are plenty of fields to look through and read the documentation about to be able to send APN messages to, as an example flow, all devices for a given user. CustomUserData is your friend there. The simplest next step is a hello world type of push alert. There are plenty of examples for that kind of thing on GitHub/google. While troubleshooting, `https://github.com/qeude/Swush` can be used to send directly, so it's a nice option to bypass Amazon SNS and see if there's something happening elsewhere in an app ecosystem.

Other Third-Party APNs Backends

Amazon SNS is a fairly simple tool. Thus the name. But like all tools, it grows and becomes more complicated as customers ask product managers to build more and more features. Thus, there are other options that are even easier to use. There are plenty (as there are plenty of third-party options for all AWS services), which include

https://www.courier.com/

https://www.magicbell.com

https://www.catapush.com

https://knock.app

https://onesignal.com

https://www.braze.com

https://firebase.google.com

https://www.airship.com

Some of these are there to provide better dashboards, a simpler experience, give more of the code and automation options for free, extend notification options to inside apps, add integrations to marketing tools, enrich what appears in notification dialogs, or just be easier to call in code. Even though there are plenty of tools that reduce friction or provide new customization options, it's also possible to build a custom backend to handle notifications from web apps as well.

Building a Custom Notification Backend

Apple provides sample code for a number of different languages to build your own if you don't want to use one of the commercial ones listed above.

To build an Apple Push Notification service (APNs) to send push notifications to an iPhone or Mac app, you will need to

1. Create an Apple Developer account.
2. Enable Push Notifications for your app. To do this, go to the Capabilities tab in Xcode and select the Push Notifications capability.

3. Generate an APNs certificate. You can do this in the Certificates, IDs & Profiles section of the Apple Developer website.
4. Create an APNs server. This can be done using any programming language or framework.

Once these steps are done, start sending push notifications to the app. Doing so requires sending a POST request (HTTP/2) to the APNs server with the device token to send the notification to and the payload of the notification in the form of a JSON object. Here is an example of a POST request that you can use to send a push notification to an iPhone:

```
POST /3/device/{device_token} HTTP/2
Host: api.push.apple.com
Authorization: Bearer {provider_token_jwt}
apns-topic: com.example.myapp
apns-push-type: alert
Content-Type: application/json
{
  "aps": {
    "alert": {
      "title": "My Notification Title",
      "body": "This is the body of my notification."
    }
  }
}
```

Once the POST request is sent, the APNs server will deliver the notification to the device provided everything is configured properly. If certificate-based authentication is used instead of token-based authentication, the provider certificate is presented for TLS auth and the Bearer token header is not used. If the device is online, delivery can happen immediately; if not, APNs may deliver later depending on notification type and expiration settings. However, keeping track of devices, topics, and everything else involved will require a fair amount of additional logic and programming. So it's usually easiest to get started with a tool like SNS which provides developers much of that for free.

No matter which tool is used to act as the delivery vehicle for push notifications, devices may still have problems receiving them, which can require a bit of spelunking to figure out exactly what's happening.

Debugging APNs

Troubleshooting push notification communications between servers and Apple's Push Notification agent on devices often starts and ends with a device that has push notifications for a given app disabled. Notifications can be disabled globally or on a per-app basis. To check global settings, open System Settings on Mac or Settings on iPhone and then check the global configuration options, making sure they match what is desired, as seen in Figure 5-19.

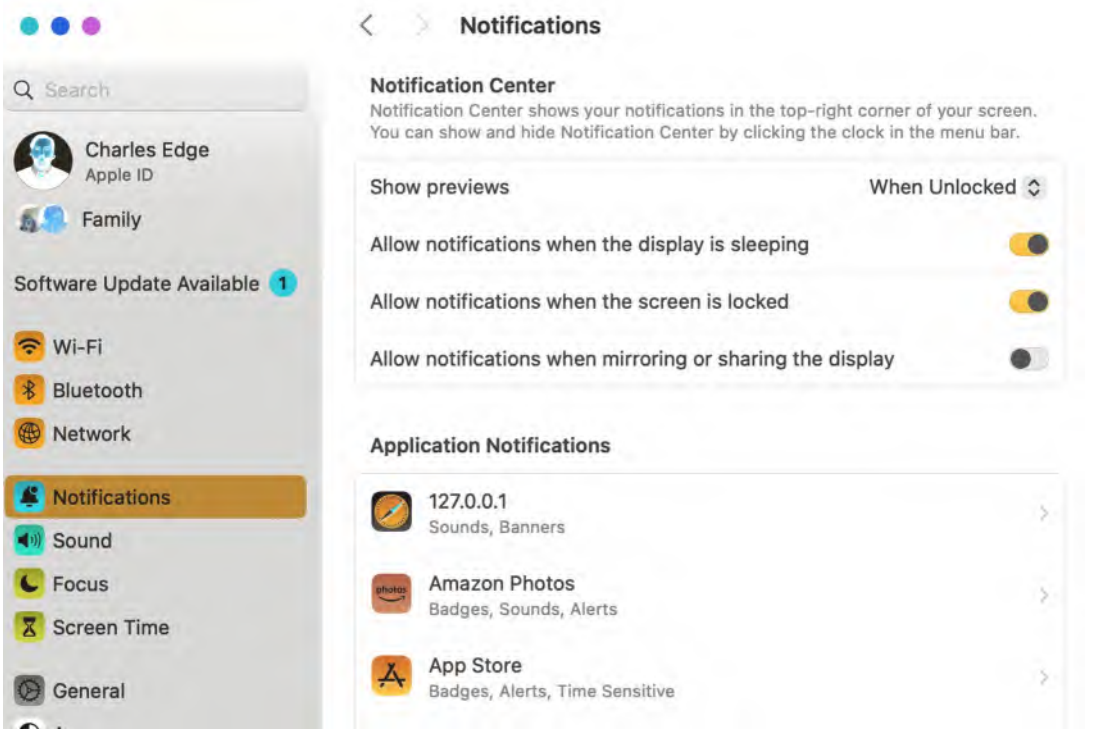

Figure 5-19. *Notification settings*

The per-app options are available by clicking on each app (or website, but let's focus on apps here) and clicking on the app to configure. Here, make sure the "Allow notifications" option is enabled and that how and when notifications for the service appear are configured in a consistent manner with what's needed, as seen in Figure 5-20.

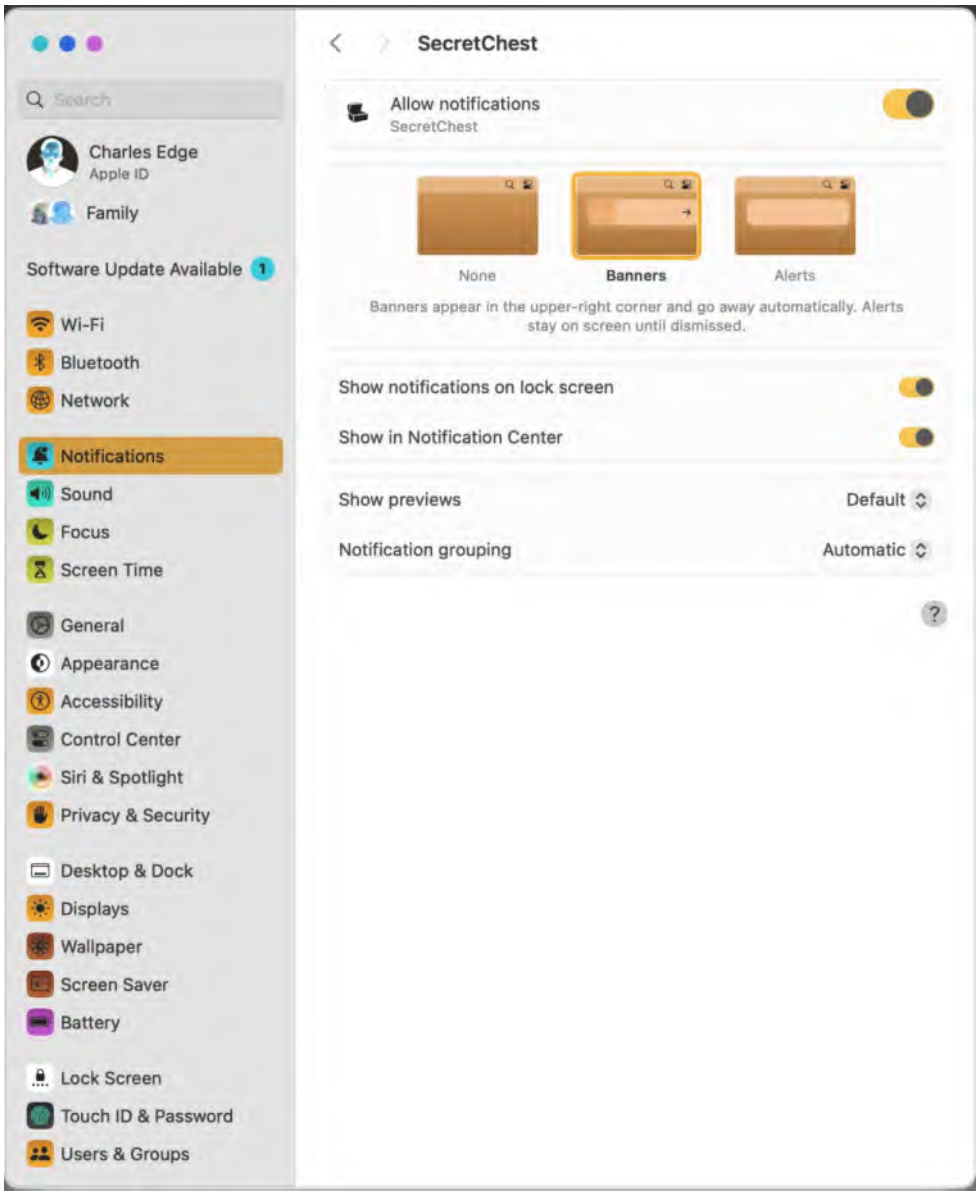

Figure 5-20. *Notification settings detail for Secret Chest*

Provided that users have notifications enabled, debugging can get really technical from there. Open the Console app and search for apsd, apns, topic, or any of the other buzzwordy-strings listed throughout this chapter (or the name or URL of the server sending out the notifications). The response can be verbose (see Figure 5-21) but will help to at least get started trying to find the source of the issue.

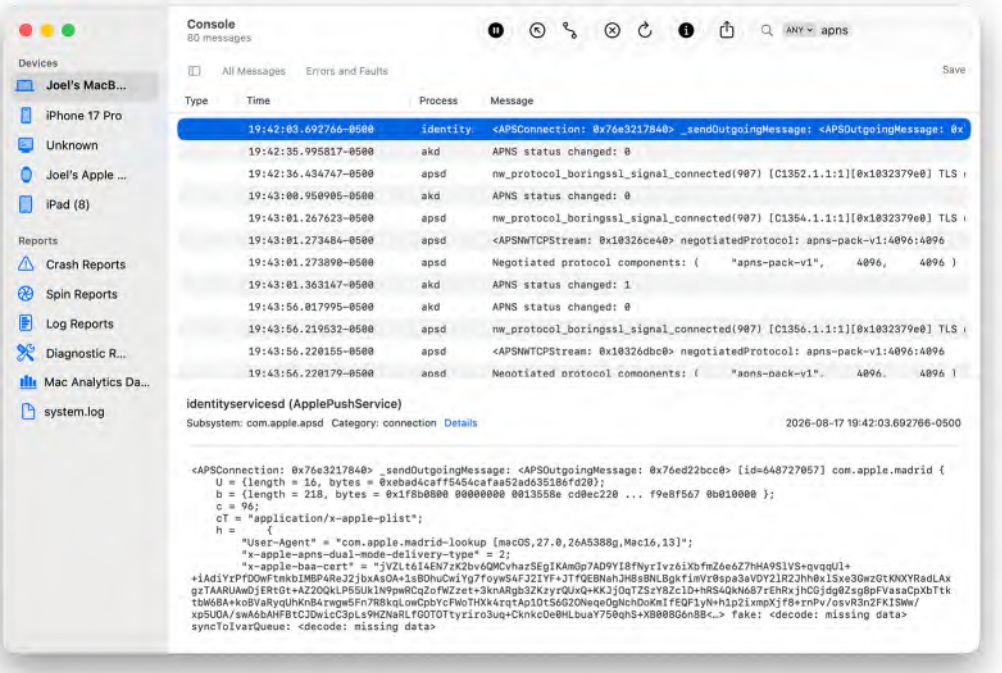

Figure 5-21. *APNS logs in Console.app*

If no entries are found on devices, or not enough information, it's also possible to put the local agent into debug mode (yes, it's agentless, but that just means the agent is built into the operating system). To do so, configure the preferences for com.apple.apsd to increase the APSLogLevel to 7 and enable APSWriteLogs, killing the daemon once configured with the following three commands (or separate them with a semicolon to merge them into a single command):

```
defaults write /Library/Preferences/com.apple.apsd APSLogLevel -int 7
defaults write /Library/Preferences/com.apple.apsd APSWriteLogs -bool TRUE
killall apsd
```

Then use tail -f to watch the apsd.log file at /Library/Logs/apsd.log.

```
/usr/bin/tail -f /Library/Logs/apsd.log
```

Be wary, as this can fill up a system. So best to try for a short span and then disable, with the following commands:

```
defaults write /Library/Preferences/com.apple.apsd APSWriteLogs -bool FALSE
defaults delete /Library/Preferences/com.apple.apsd APSLogLevel
killall apsd
```

Another option if devices aren't getting any push notifications is to attempt to connect directly to ports. TwoCanoes makes an app to do this called Push Diagnostics, available on the App Store at https://apps.apple.com/us/app/push-diagnostics/id689859502?mt=12. Once installed, click the Start button to start testing each services availability from the device, as seen in Figure 5-22.

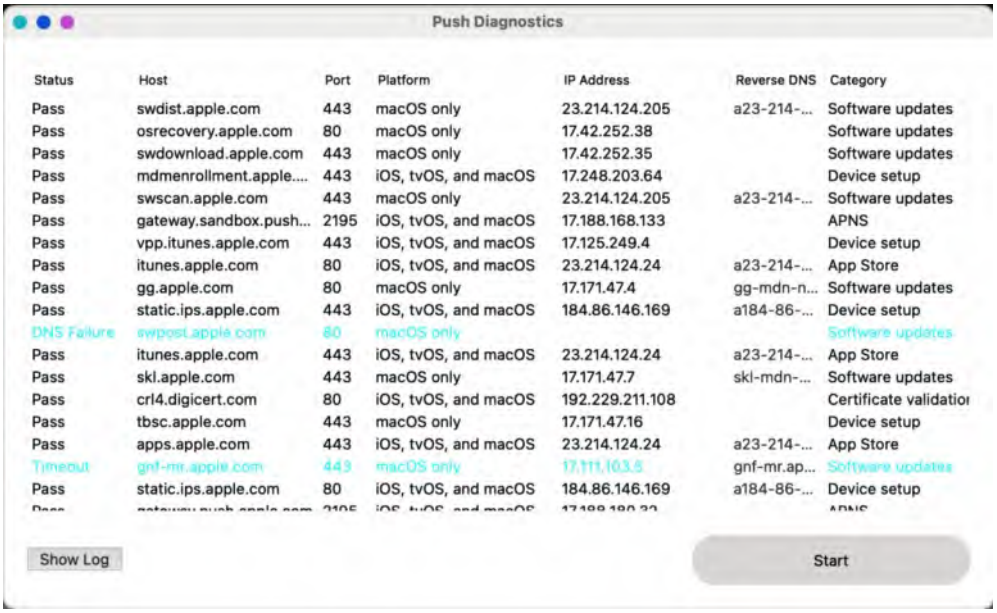

Figure 5-22. *Push Diagnostics in use*

When finished, click the Show Log button to see raw connection logs, as seen in Figure 5-23.

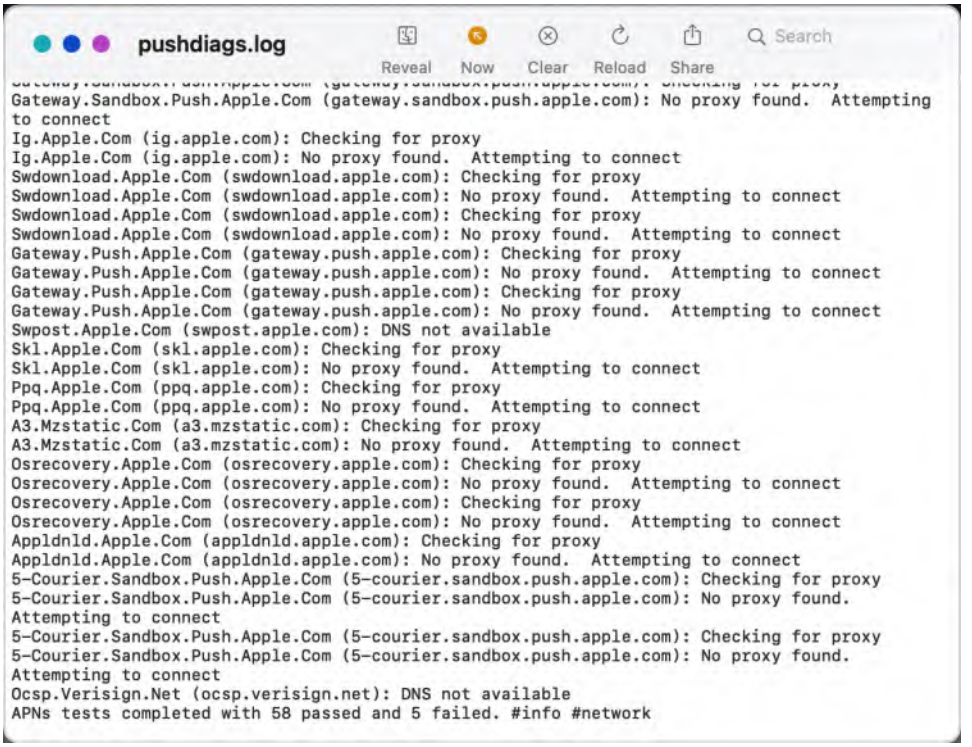

Figure 5-23. *Logs in Push Diagnostics*

If a port can't connect, try it again from another network. If a firewall or other device on a network doesn't allow access to these ports or hosts, push notifications just aren't going to work. It's also possible to do much of this from the command line as well. Use netcat to attempt to establish connections with Apple's push API host. That happens over port 443 (or 2197 for APNs traffic), and it can be a bit laggy, so a -w to timeout will cut out all the cruft (which is at Akamai so that shouldn't be too much of a problem). That command would look like the following:

```
/usr/bin/nc -v -w 15 api.push.apple.com 443
```

Another option is to add a -4 to force connections over IPv4 and check the APNs development host over port 443:

```
/usr/bin/nc -v -4 api.development.push.apple.com 443
```

It's also possible to add a -z option, to just scan for daemons, without actually sending any data Apple's way:

```
/usr/bin/nc -vz -4 api.push.apple.com 443
```

Because of how NAT works, notice that the src port keeps changing (incrementing actually). It's possible go ahead and force the source port to stay the same as the destination port using the -p option, which helps find other firewall or gateway configuration problems:

```
/usr/bin/nc -vz -4 -p 2197 api.push.apple.com 2197
```

It's also possible to test that it's what's inside the packets rather than the raw ports that are problematic. This can help triangulate if the problem is with something like a stateful packet inspecting firewall blocking information it can't read because of something like certificate pinning. To do so, set up a custom listener on a device outside the network – or start on the local subnet, then move to another subnet on the same network, and ultimately to another network in order to check zone-by-zone, so-to-speak, for a failure. To create a listener with netcat in a few seconds, use the -l option on another host in the zone to check:

```
/usr/bin/nc -l 2197
```

Then scan localhost to make sure the blocker isn't another daemon (e.g., an antivirus or security tool):

```
/usr/bin/nc 127.0.0.1 2197
```

It's also possible to do this on a range of ports:

```
/usr/bin/nc 127.0.0.1 2197-2198
```

Now, as is often the case, if the connection problem is because data isn't parodying, use nc to check that using the -x operator followed by an IP and then : and a port. For example:

```
/usr/bin/nc -vz -4 -w 10 -p 2197 -x 10.0.0.2:8080 api.push.apple.com 2197
```

There are plenty of other issues that can crop up with push notifications, but these steps should get close to the answer, if not pinpoint a configuration or connectivity problem. Once the connections can be established, it's time to flow data into the app via that push notification. The most common way is to deep link into an app.

Deep Linking into Apps

One of the most powerful aspects of notifications is that users can be directed to a specific view in an app. The message that users receive can open a view and even trigger an option or a change to a json file, which then loads content into the app – or better tells the app to come get the information from a web service, since what's in a push notification shouldn't be considered safe. So for example, don't send a password or a private key in a message, but send a command to the app to check the web service to get the secret.

Deep linking works by using a special URL scheme that is associated with an app. When a user clicks on a link that uses this URL scheme, their device will open the app and open the specific part of the app that is associated with the URL scheme. To add deep linking to a swift app, the following steps are required:

- Create a URL scheme for the app.
- Add the URL scheme to the info.plist file in the app.
- Implement the openURL() method in the app delegate.
- Create a URL scheme for the app.
- The app must be installed for the URL scheme to be available.

The first step is to create a URL scheme for the app. A URL scheme is a unique identifier that is used to identify the app. This is no different than how http:// opens a web browser or ftp:// opens an FTP server, or gopher:// opens a gopher server. Mostly because the creators of those protocols helped define the structure of URIs in RFC 1738, available at `https://datatracker.ietf.org/doc/html/rfc1738`. Less has changed since then than anyone might think, except that it's common for an app, like the Finder, to tap an event like file:// and then take an action when that URI is encountered.

When these are being sent over the web, like through a push notification, they're usually thought of as URLs instead of URIs. To create a URL scheme, you need to use the following format:

```
appname://
```

For example, if the app's name is "MyApp", then the URL scheme would be "myapp://". The next step is to add the URL scheme to the app's Info.plist file. To do so, open the app's Info.plist file and add the following key-value pair:

```
key: CFBundleURLTypes
value:
  - CFBundleURLSchemes:
    - myapp
```

The final step is to implement the openURL() method in the app delegate. This method will be called when a user clicks a link that uses the app's URL scheme. Here is an example of how to implement the openURL() method:

```
func application(_ application: UIApplication, open url: URL, options:
[UIApplication.OpenURLOptionsKey: Any]) -> Bool {
if url.scheme == "myapp" {
// Handle the deep link
return true
}
return false
}
```

This code will handle the deep link if the URL's scheme is equal to "myapp". Other uses for deep linking, like callbacks, are covered in the next chapter. Next, though, let's look at result builders.

Result Builders

Result builders are used to create custom DSLs (domain-specific languages). A DSL is a programming language that is specialized for a particular domain, such as UI development, web development, or data processing. Result builders can be used to create DSLs that are more expressive and easier to use than swift for specific tasks.

To build a result builder, create a struct or enum with the `@resultBuilder` attribute. This requires implementing build methods (such as `buildBlock(_:)`) that are used to build the result of the result builder.

Here is an example of a simple result builder:

```
@resultBuilder
struct StringBuilder {
  static func buildBlock(_ components: String...) -> String {
    return components.joined(separator: "")
```

```
  }
}

func buildGreeting(@StringBuilder _ builder: () -> String) -> String {
  return builder()
}
```

This result builder can be used to build strings in a more expressive way. For example, the following code breaks down a greeting into its parts:

```
let greeting = buildGreeting {
  "Hello, "
  "world!"
}
```

This is equivalent to the following code:

```
let greeting = "Hello, world!"
```

However, the StringBuilder result builder is more expressive because it allows developers to break up the string into multiple lines.

This is also common with callbacks and structures in URLs. URLs aren't just to call services on a local host, they're also used to call services that live outside the device, such as swift-based microservices hosted in a service like AWS Lambda or with a Google Cloud Function.

Building Microservices in Swift

Most modern apps also need a hosted component somewhere out there on the web. This usually meant developers had to bring in people from different disciplines to get the web services needed built. The swift apps could then consume them. However, some of the cloud providers now support hosting microservices built in swift (that's one of the many reasons the language itself is an open source project).

These microservices are the backbone of what many call "serverless" today. AWS Lambda provides a serverless computing environment that allows developers to execute code in response to events without provisioning or managing servers. While AWS Lambda primarily supports languages like JavaScript, Python, and Node.js, with the introduction of the swift AWS Lambda Runtime, developers can leverage swift scripts as Lambda functions.

Being able to use swift provides safety, performance, and expressiveness to make it an attractive language for server-side development. Using swift on AWS Lambda also allows developers to harness existing swift skills and ecosystems to build serverless applications. Serverless computing then eliminates the need for server management and scaling, allowing developers to focus on writing code and delivering value. This also paves the way for using other AWS services like API Gateway, S3, DynamoDB, SNS (as described earlier in the chapter), and others to enable the creation of scalable and event-driven serverless architectures.

To get started, first install the Swift AWS Lambda Runtime, which provides the necessary libraries and tools for running a Lambda function written in swift. To get started, make sure there's an AWS account at `https://aws.amazon.com` and create an IAM role with the required permissions for a Lambda function to interact with other AWS services (if access to other services is indeed required). Once logged in, click the Services menu at the top of the screen and choose AWS Lamda. Provided the account is set up correctly, there will be a button to Create Function, seen in Figure 5-24.

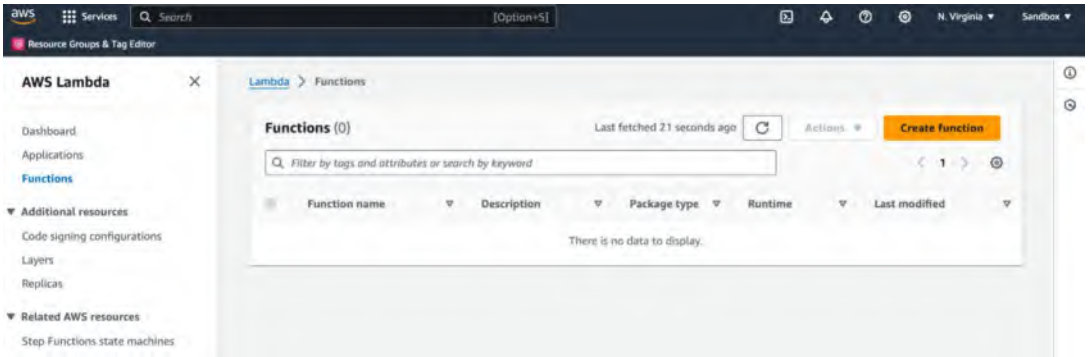

Figure 5-24. *Creating a Lambda Function in AWS*

Get the latest Swift AWS Lambda Runtime library from the GitHub repository `https://github.com/awslabs/swift-aws-lambda-runtime` and install it so it can be imported into a project. Then, open Xcode and create a new project that imports the runtime, as follows:

```
import AWSLambdaRuntime
// My code's logic
func handle(_ event: Event, context: LambdaContext) async throws ->
Output {
  // ...
}
```

For the purpose of this sample script, let's look at a function that just returns the current time in epoch:

```
exports.handler = async (event) => {
  // Get the current UTC time in milliseconds
  const currentTimeInEpoch = Math.floor(Date.now() / 1000);
  // Create the response object
  const response = {
    statusCode: 200,
    body: JSON.stringify({
      message: 'Current time in epoch:',
      data: currentTimeInEpoch,
    }),
  };
  // Return the response
  return response;
};
```

To step through the above code, it creates a handler function (exports.handler) to define the function that will be invoked by AWS Lambda. The async (event) => { ... } makes the function asynchronous, allowing for await operations. The const currentTimeInEpoch = Math.floor(Date.now() / 1000); gets the current UTC time in milliseconds using Date.now() and converts it to seconds (epoch) by dividing by 1000. The const response = { ... } creates a response object with a statusCode set to 200 for successful execution and then responds with a body containing a JSON string with a message and the calculated epoch time to the return response object, sent back to the invoking service.

Once there's a function, package the executable along with the swift standard library and AWS Lambda runtime in a deployment package (ZIP file). Make sure it can run within a few minutes or the Lambda will fail. There is a Timeout field that indicates the maximum amount of time in seconds that a function can run in AWS. The default is 3 seconds, but it can be adjusted to up to 15 minutes (although it gets expensive the longer it runs, so be aware of how efficient code is.

At the new function screen, give the function a name and select the Amazon Linux runtime option, which allows for custom runtimes (Figure 5-25). Click the Create Function once the correct options are selected.

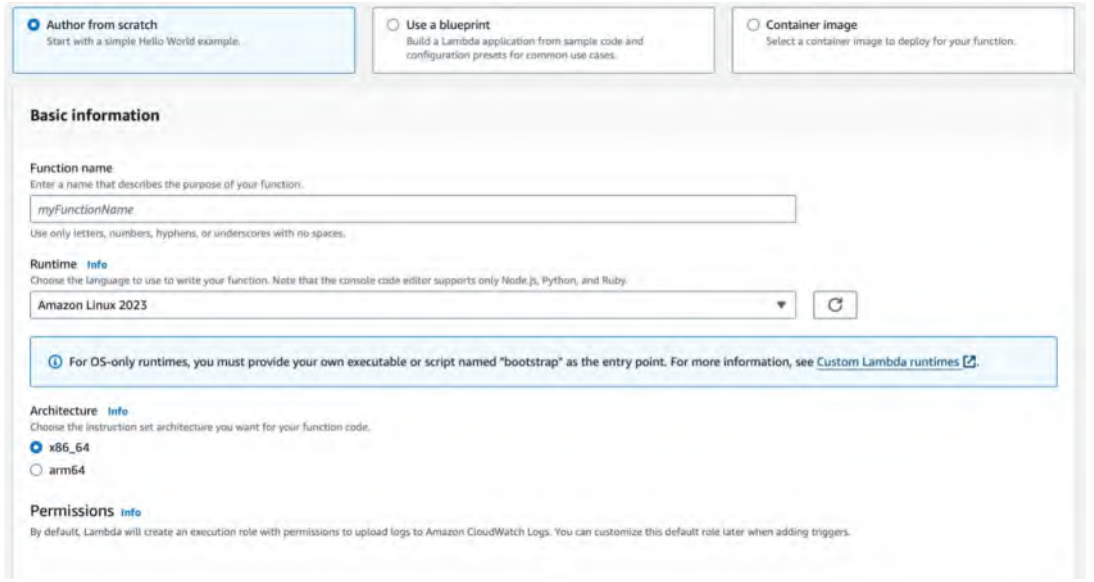

Figure 5-25. *Configuring the runtime for a Lambda in AWS*

At the screen to set up the new function, scroll down to the code section and select the .zip option using the Upload From field (Figure 5-26).

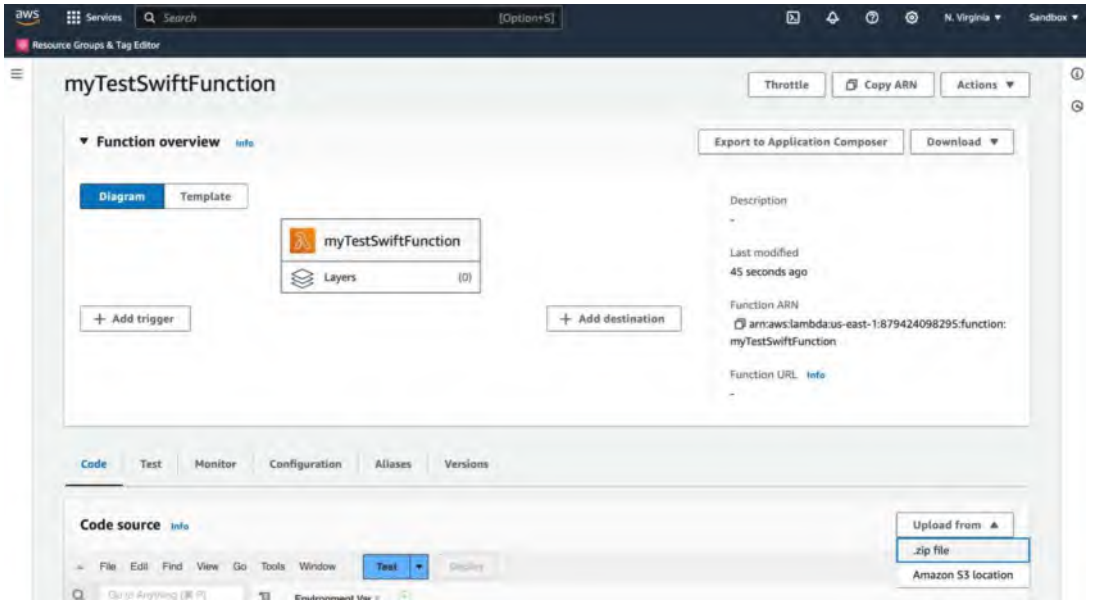

Figure 5-26. *Upload a zip file to create a Lambda in AWS*

Upload the deployment package to Lambda and configure the function to use a custom runtime (e.g., `provided.al2`/`provided.al2023`). The AWS Lambda Runtime still needs to be included in the Swift Package Manager configuration for the project. Now it's possible to invoke the Lambda function to run the code.

The third-party runtime library provides support for swift, but AWS does not provide official support for it (or any third-party libraries as that would be practically impossible. For the most up-to-date information on supported runtimes, check the AWS Lambda documentation.

The new Lambda function can then be tested locally using tools like the AWS SAM CLI or the Swift AWS Lambda Runtime's local testing capabilities and triggering programmatically, either using a URL, by various AWS SDKs, and frameworks. For example, the AWS SDK for Swift exposes features from Amazon S3, DynamoDB, or API Gateway.

Amazon isn't the only place to load and run serverless code. There are others, like Google. Google Cloud Functions don't directly support swift as a runtime language, but as done with Amazon, there are a couple of alternative approaches to achieve server-side Swift functionality on Google Cloud. The first is to use a server-side swift framework like Vapor, Kitura, or Perfect. This involves a Google Cloud Project and installing the Google Cloud SDK, using docker to build and package swift apps as docker images (also an option with Amazon). Google can also run code using the Cloud Run service, which is generally simpler and more efficient.

The process is similar with Alibaba, Knative, Cloudflare, Kubeless, etc. On Microsoft's Azure, Azure Functions does not provide a first-party Swift runtime, so Swift discussions commonly focus on approaches like these:

- Classic worker: Azure Functions does not provide first-party Swift support for a classic worker runtime.
- Custom handler: This option allows you to define handlers for different event triggers and implement your own logic in Swift.

Installing the Azure Functions Core Tools is as easy as using npm to do so:

```
npm install -g azure-functions-core-tools
```

Creating a new Azure Functions project for a custom handler is then easily automated, as follows:

```
func init SwiftFunction --worker-runtime custom
```

It's worth noting that AWS is probably the most popular serverless option currently in use, and so there will be more community support when doing so. However, there are plenty of other options, and who knows how long their dominance will last, especially when vendors like Microsoft actually make it easier to use swift on their platform.

Conclusion

Notifications and data flow are two halves of the same problem: getting information to a user at the right moment, and getting it to the right place inside the app once they act on it. The chapter followed that path end to end. Local notifications are handled entirely within an app, while remote notifications travel through APNs, and the delivery flow behind them is always the same handful of steps - ask for permission, receive a device token, hand that token to a backend, send a payload to APNs, and let Apple deliver it to the device. Almost everything that goes wrong with push notifications goes wrong somewhere in that chain, which is why so much of the setup work involves certificates, keys, topics, and entitlements rather than code.

There's no single right way to run the provider side of that chain. Amazon SNS is a reasonable starting point and the certificate and provisioning work covered here applies no matter which service sits behind it. Services like Courier, MagicBell, OneSignal, Braze, and Firebase trade some control for better dashboards, easier integrations, and less code to maintain. Rolling a custom backend using HTTP/2 POST requests to api.push.apple.com is entirely doable and gives complete control, but it also means owning device token storage, topic management, retries, and expiration logic that the hosted options provide for free. Whichever route is chosen, expect to spend real time debugging - and remember that the most common cause of a missing notification isn't the backend at all, it's notifications being disabled globally or for that specific app on the device.

The back half of the chapter covered what happens after a notification arrives. Deep linking through a custom URL scheme lets a notification open a specific view rather than dumping the user at a launch screen, with the important caveat that a payload should carry a command to go fetch something, never the secret itself. Result builders show how Swift can be shaped into a domain-specific language, which is the same machinery that makes SwiftUI read the way it does. And microservices in Swift - on AWS Lambda,

Cloud Run, Azure Functions custom handlers, or a Vapor container almost anywhere - mean the backend that feeds those notifications can be written in the same language as the app that consumes them. That's a relatively new luxury, and it collapses a lot of the distance between client and server work that used to require separate teams.

In the next chapter, we'll look at asynchronous programming in Swift, including async/await, task coordination, and practical patterns for handling concurrent work.

CHAPTER 6

Asynchronous Programming

SwiftUI is, or at least can be, declarative. Declarative programming is a programming paradigm that focuses on expressing the logic of a computation without explicitly stating the control flow. In other words, tell the program to achieve rather than how to achieve it step by step. The ability to achieve the desired state is then either wrapped up into an API and obfuscated from the developer, or defined by the developer so desired states can be declared.

In other words, declarative programming focuses on the "what" or the desired outcome or results the program is to produce. The programmer doesn't specify the exact steps or algorithm for execution. Instead, declarative languages handle the "how" a state is achieved. The language's implementation or runtime system takes care of figuring out the necessary steps to achieve the desired outcome. This allows declarative code often resembles natural language, making it easier to understand and maintain.

The declarative style can be used within imperative languages: Some imperative languages like Python and JavaScript support declarative features, such as list comprehensions and functional. Tools like Ansible, Chef, and Puppet use declarative languages to describe system configurations, as does Apple's more recent implementation of Declarative Device Management, or DDM. Declarative frameworks like React, Vue, Angular, and to some degree SwiftUI then allow developers to create user interfaces by describing their desired state.

Declarative paradigms in computing lend themselves well to asynchronous programming. Asynchronous programming is a programming paradigm multiple tasks can be run at the same time, without having to wait for each task to finish before starting the next one. This can improve the performance and responsiveness of apps and make them more user-friendly.

C. Edge et al., *Beyond Scripting*, https://doi.org/10.1007/979-8-8688-3035-8_6

In synchronous programming, each task is executed one after the other, in a linear order. This means that if a task takes a long time to complete, the entire program is blocked until that task completes. With asynchronous programming, tasks can be interleaved and may run in parallel, so the program runs even if a task takes a long time to complete. A common way to program asynchronously is to use callbacks. A callback is a function that is called when a task is finished. When a task starts asynchronously, it's passed a callback function to the task. The task will then call the callback function when it is finished.

Another common way to implement asynchronous programming is to use promises (mentioned earlier). A promise is a special object that represents the result of a task. When a task is started asynchronously, it produces a promise for the result of the task. The promise will be resolved when the task is finished, and it will contain the result of the task. Asynchronous programming then can improve the performance of programs by allowing multiple tasks to run at the same time. This can be especially beneficial for tasks that take a long time to complete, such as network requests or file IO. Programs can also be more responsive, as users can interact with the program even while long-running tasks are still running in the background.

Asynchronous programming does have its challenges. It can be more complex than synchronous programming, as developers have to keep track of multiple tasks and their dependencies. Error handling can be more complex, as errors have to be handled when they show up in any of the tasks. Debugging asynchronous programs can also be more difficult than debugging synchronous programs, the execution of multiple tasks has to be tracked manually. But again, one of the most common means to implement asynchronous programming in apps is to use callbacks.

This chapter follows a practical roadmap:

1. Callback-based patterns
2. Concurrency with actors and threads
3. Modern async/await and Combine approaches
4. Networking sync patterns for web services (AsyncHTTPClient, webhooks, polling)
5. Memory risks like object bloat/retain cycles and ways to reduce them

Callbacks

As mentioned earlier, callbacks are a common, traditional approach to manage tasks asynchronously. These are commonly used in web programming, especially when there's a dependency on a task, like with OIDC mentioned earlier in the book – in that case, a user authenticates and then there's a callback URL to be accessed once the authentication is complete.

A callback is a function that is passed as an argument to another function. The first function, called the caller, calls the callback function when it is finished. The callback function, called the callee, can then do whatever it needs to do, such as update the user interface or perform some other operation. Here is an example of a callback:

```
func downloadImage(
  url: URL,
  completion: @escaping (UIImage?) -> Void
) {
  URLSession.shared.dataTask(with: url) { data, _, error in
    guard error == nil, let data = data else {
      completion(nil)
      return
    }
    let image = UIImage(data: data)
    completion(image)
  }.resume()
}
```

In this example, the downloadImage function takes a URL as an argument and a callback function as a parameter. The callback function is called when the image has been downloaded. The callback function takes a single argument, which is the image that was downloaded. To use the downloadImage function, do something like this:

```
let url = URL(string: "https://example.com/image.png")!
downloadImage(url: url) { image in
  if let image = image {
    // Do something with the image
  }
}
```

In the above example, the callback function is called when the image has been downloaded. The callback function checks if the image is not nil, and if it is not nil, the callback function does something with the image. Given that a number of tasks can run concurrently, it's then important to understand the impact concurrency has on actors and threads.

If that callback updates UI state, dispatch the update to the main thread (e.g., `DispatchQueue.main.async { ... }`) so view updates remain safe and predictable after network operations.

Managing Concurrency with Actors and Threads

Think of callbacks as a means to run a series of tasks that's separate from another series of tasks in parallel. However, another approach is to think of code through a lens of concurrency, where a declarative interface is able to more intelligently derive the state of objects. Swift provides two main ways to manage concurrency: actors and threads. Actors are a new concurrency feature that was introduced in Swift 5.5 that provide a safe and efficient way to manage shared mutable states. Threads are a more traditional concurrency mechanism that allow for running multiple tasks in parallel.

Actors are a type of object that can be accessed concurrently from multiple threads, but ensure that access to their mutable state is serialized and thread-safe. This provides assurances that two threads will never try to access the same actor's mutable state at the same time, which can prevent data races and other concurrency problems. To create an actor, use the actor keyword. For example:

```
actor MyActor {
  var value = 0
  func increment() {
    value += 1
  }
}
```

Once created, actors are accessed from multiple tasks. For example:

```
let actor = MyActor()
Task {
  await actor.increment()
```

```
  print(await actor.value)
}
Task {
  await actor.increment()
  print(await actor.value)
}

// Console will print "1" then "2"
```

The two Tasks run in parallel and could finish in either order.

Threads are a more traditional concurrency mechanism that allow for multiple tasks to run in parallel. A common way to schedule concurrent work is to use a DispatchQueue and call its async method. For example:

```
DispatchQueue.global().async {
  // Do something on a background thread
}
```

When work is scheduled, specify the queue that the work will run on. This allows for controlling where the work will run and how it will interact with other threads. It can be like the wild west if some simple guidelines aren't followed. Use actors to protect shared mutable state from concurrent access. Use threads to run multiple tasks in parallel, but when there's no need to protect shared mutable states. In general, actors are a good choice for tasks that involve shared mutable states. For example, a class that maintains a list of items could use an actor to ensure that the list is always thread-safe.

Threads are a good choice for tasks that don't involve shared mutable state. For example, if you need to download a file from the internet, you could use a thread to download the file in the background. All of this allows a program to stay in sync with another service, like a web service.

Asynchronous Options in Swift

Swift offers several built-in options for asynchronous programming, allowing for code that handles long-running tasks efficiently without blocking the main thread. The main options most will use include closures, async/await, and frameworks like Combine.

Closure-Based Asynchronous Patterns use a completionHandle parameter to pass a closure as a parameter to a function to execute code after an asynchronous operation completes. This is most commonly used in networking and heavier file I/O functions like syncing files with iCloudKit. An example would be

```
func fetchData(completionHandler: (Data?, Error?) -> Void) {
    // Perform asynchronous task...
    let data = ...
    completionHandler(data, nil)
}
```

The async and await keywords, introduced in Swift 5.5, allow asynchronous code to be in a more sequential style by using async to mark a function as asynchronous and then await to pause execution within an asynchronous function until a task completes. Since these usually come in pairs, they are colloquially simply called async-await:

```
func fetchData() async throws -> Data {
    // Perform asynchronous task...
    let data = ...
    return data
}
```

Alternatively, the Combine framework can be used. It's not a part of the Swift standard library, but is easily imported and probably the most declarative approach to asynchronous programming as it exposes publishers, operators, and subscribers to manage asynchronous events and data streams. To show this in action, consider the following code:

```
let dataPublisher = URLSession.shared.dataTaskPublisher(for: url)
    .map(\.data)
    .eraseToAnyPublisher()
```

Combine also provides the `Future` type. This represents the eventual result of an asynchronous operation and can be chained together and combined with the map, flatMap, and other methods. An example of a promise and future combination would be something like the following, swapping out the commented out section with an actual task, as follows:

```
import Combine
```

```
func fetchData() -> Future<Data, Error> {
    // Perform asynchronous task...
    let data = ...
    return Future { promise in
       promise(.success(data))
    }
}
```

There's no absolutes right approach as complexity and familiarity vary wildly in developers and code. However, closure-based patterns are simpler for basic tasks while async/await offers a more structured approach and the combine framework excels in complex declarative and so reactive scenarios. Whichever option is selected, consider error handling carefully: ensure proper error propagation and handling in asynchronous code to avoid unexpected crashes.

When running into unexplained issues while you're coding, it's a good idea to check the asynchronous parts of your code first. It's not uncommon to have a function return before all of the asynchronous tasks were completed, for example, if you're not careful.

Asynchronicity with Web Services

The idea of asynchronous programming was popularized by the introduction of Ajax for Google Apps. It was amazing when it came along, to see a change, like an email that got deleted from a mail application to then disappear on a screen without a refresh. That saved Google plenty of resources on servers because less monolithic junk was being loaded into a browser. Therefore, it's no surprise that programmers who worked in any language quickly wanted to keep interfaces in sync between a web interface and an app.

There are a number of different ways to keep a declarative interface in sync with a web service. One common way is to use a versioning system. A versioning system allows tracking changes to code and data and to roll back changes if necessary.

AsyncHTTPClient

AsyncHTTPClient (AHC) is a Swift library that enables asynchronous HTTP requests and responses, meaning an application can continue with other tasks while waiting for network operations to complete. It's particularly useful for efficient handling of I/O-bound operations, improving responsiveness and preventing blocking threads.

Here's a breakdown of AHC in Swift:

AHC in Swift is available at https://github.com/swift-server/async-http-client and built on SwiftNIO. It features concurrency support, asynchronous and non-blocking requests, a streaming body download, TLS support, automatic HTTP/2 over HTTPS, and cookie parsing.

AHC allows developers to initiate HTTP requests without blocking the current thread, enabling applications to perform other actions while waiting for responses. AHC manages a pool of open connections to servers, improving performance by reusing connections for subsequent requests. Request timeouts then prevent indefinite waiting for responses. Large response bodies can be streamed, reducing memory usage.

To use AsyncHTTPClient, first add the package dependency to the Package.swift file:

```
dependencies: [
    .package(url: "https://github.com/swift-server/async-http-client.git",
    from: "1.10.0")
]
```

Then import it into code:

```
import Foundation
import AsyncHTTPClient
```

Next, create an HTTPClient instance:

```
let httpClient = HTTPClient(eventLoopGroupProvider: .createNew)
```

Once there's an instance, make a request:

```
func fetchData() async throws {
    let request = try HTTPClient.Request(url: "https://api.example.
    com/data")
    let response = try await httpClient.execute(request: request)
    if var body = response.body {
        let bytes = body.readBytes(length: body.readableBytes) ?? []
        let data = Data(bytes)
        print(String(data: data, encoding: .utf8) ?? "")
    }
}
```

Once the request has been made, handle the response. This includes accessing the response status code, headers, and body – then processing the body data as needed (e.g., a standard step is to decode any JSON returned). Once the response has been processed, shutdown the HTTPClient:

```
// When finished:
try httpClient.syncShutdown()
```

AHC seamlessly integrates with async/await for modern concurrency. It handles asynchronous requests, so doesn't block I/O for efficient network operations. It can then process large responses without loading everything into memory and can do so over secure connections for HTTPS and performs automatic HTTP/2 over HTTPS. AHC can also handle redirects. Another way that events are handled is with webhooks.

Webhooks

Another common way to keep a declarative Swift interface in sync with a web service is to use webhooks. Webhooks are a way to receive notifications when changes are made to a web service. Webhooks can be used to trigger an update to a declarative interface whenever the underlying web service changes. For example:

```
import Foundation
func callWebhook(url: URL, data: Data) {
  // Create a URLSession
  let session = URLSession.shared
  // Create a POST request
  var request = URLRequest(url: url)
  request.httpMethod = "POST"
  request.httpBody = data
  // Make the request
  let task = session.dataTask(with: request) { data, response, error in
    // Check for errors
    if let error = error {
      print(error)
      return
    }
    // Check for the response
```

```
    if let response = response as? HTTPURLResponse {
      if response.statusCode != 200 {
        print(response)
        return
      }
    }
    // The webhook was called successfully
    print("Webhook called successfully")
  }
  // Start the task
  task.resume()
}
func main() {
  // Create the URL
  let url = URL(string: "https://example.com/webhook")!
  // Create the data
  let data = "This is the data that will be sent to the webhook".
  data(using: .utf8)!
  // Call the webhook
  callWebhook(url: url, data: data)
}
main()
```

In the above example, the callWebhook function takes a URL and data as arguments. The function then creates a POST request and sends the data to the webhook. The function also checks for errors and prints the response. The main function creates the URL and data, and then calls the callWebhook function. The callWebhook function will call the webhook and then print the response. Webhooks are reactive, but polling is a bit more proactive.

Polling

Polling is another technique to keep a declarative interface in sync with a web service. Polling is the process of periodically checking the web service for changes. Use polling to update a declarative interface whenever the underlying web service changes. An example of polling:

```
import Foundation
func pollRestEndpoint(url: URL, completion: @escaping (Data?) -> Void) {
  Timer.scheduledTimer(withTimeInterval: 30, repeats: true) { _ in
    // Create a URLSession
    let session = URLSession.shared
    // Create a data task
    let task = session.dataTask(with: url) { data, response, error in
      // Check for errors
      if let error = error {
        completion(nil)
        return
      }
      // Check for the response
      if let response = response as? HTTPURLResponse {
        if response.statusCode != 200 {
          completion(nil)
          return
        }
      }
      // Return the data
      completion(data)
    }
    // Start the data task
    task.resume()
  }
}
func main() {
  // Create the URL
  let url = URL(string: "https://example.com/api/v1/data")!
  // Poll the REST endpoint
  pollRestEndpoint(url: url) { data in
    // Do something with the data
  }
  RunLoop.main.run()
}
main()
```

The above code example shows how to use polling to periodically check a REST endpoint for changes. The pollRestEndpoint function takes a URL as an argument and a callback function as a parameter. The callback function is called when the REST endpoint has been polled. The callback function takes a single argument, which is the data that was returned from the REST endpoint. The main function creates the URL and calls the pollRestEndpoint function. The pollRestEndpoint function will periodically check the REST endpoint for changes. After each poll, the callback function will be called with the current data.

The best way to keep a declarative interface in sync with a web service will depend on the specific requirements of your application. However, versioning systems, webhooks, and polling are all common and effective ways to do so. Whichever means is chosen, use a consistent naming convention for your data models and views. This will make it easier to identify the changes that need to be made when the underlying web service changes. Keep in mind to use webhooks to trigger an update to an interface whenever the underlying web service changes and poll the web service periodically to check for changes.

Object Substantiation

Object substantiation is the process of creating new objects in memory. It's crucial to understand how object creation and reference relationships can lead to memory leaks. This is covered throughout the book, especially when discussing debugging and code quality. However, with declarative and asynchronous concepts, it's easy to create an explosion of new objects. Memory leaks can still happen in Swift, especially when retain cycles are introduced.

As mentioned, ARC helps manage memory, but it does not prevent all memory leaks. A retain cycle occurs when two or more objects hold strong references to each other, preventing ARC from deallocating them. This often happens when a closure captures self strongly. Forgetting to release references to objects when they're no longer needed can also lead to leaks. Not removing observers from notification centers can keep objects alive unnecessarily. All of this can become common when dealing with so many closures and potentially crashed bits of code. Therefore, an option is to break retain cycles with weak or unowned references. To do so, use weak references when the object might become nil or use unowned references when you're certain the object will always exist. For example:

```
class MyClass {
    var delegate: MyDelegate?
    func doSomething() {
        someClosure { [weak self] in
            self?.delegate?.someMethod()
        }
    }
}
```

Another option is to nil out references when done with an object. To do so, set properties to nil when they are no longer needed, remove objects from arrays or dictionaries when appropriate, and remove notification observers when possible. Use the removeObserver method to unregister observers when they're no longer needed. Structs and enums don't create reference cycles, so use them when possible.

Finally, Xcode's Instruments tool has features like the Leaks profiler to identify leaks. But it's worth a larger discussion on object bloat and how to correct that.

Object Bloat and Amelioration

Just like packing too much junk in a backpack, bulky objects in code can weigh it down, impacting performance and maintainability. Object bloat describes an object, or collection of objects that grows unnecessarily large and complex, often holding data and functionality it doesn't inherently need (but might have inherited). Think of a Swiss Army knife trying to be a toolbox – cluttered and inefficient.

This bloat manifests in several ways. One is from excessive properties. Packing an object with a kitchen sink of properties, especially when some are rarely used, adds unnecessary memory overhead. Methods can also sprawl, with every object handling tasks unrelated to its core purpose, which leads to spaghetti code and difficulty reasoning about logic. Nested dependencies are objects that rely on a chain of other objects to accomplish simple tasks creates coupling and hinders testing.

Object bloat isn't just an aesthetic concern; it has tangible consequences. Bulky objects take longer to initialize, allocate memory, and navigate, slowing down an app. Code also becomes convoluted, making it harder to understand, update, and debug. Bloated objects are tightly coupled, making unit testing a frustrating ordeal – one that often takes far longer than it should. This happens for a variety of reasons, like model

over-engineering, where models get stuffed with unrelated data like formatting logic or network calls that become behemoths. Sometimes classes try to do everything and so end up doing nothing well, becoming a tangled mess. Dependencies also scope creep – or objects pull in entire frameworks for small tasks, bloating memory footprint and compile times.

To resolve object bloat, embrace single responsibility; focus each object on a single, well-defined purpose so objects are small and agile. Extract complex logic into separate classes or functions, promoting clarity and reusability. Inject specific dependencies instead of monolithic frameworks, reducing unnecessary baggage. Leverage protocols and extensions: define common behavior through protocols and extend existing objects instead of creating bloat-inducing subclasses. Value types can be used when data doesn't need object behavior – consider lightweight value types for efficient memory management. Also, a common theme in this book is to embrace functional programming so functions can simplify logic and avoid unnecessary state within objects.

There are more advanced techniques as well. Use performance profiling tools to identify bloated objects and measure the impact of your refactoring efforts. Write unit tests for refactored code to ensure functionality remains intact. Remember, smaller is often better. Keep objects focused, modular, and dependency-aware. By applying these principles, code will be cleaner and more performant. Armed with smaller and more efficient code, it's time to move on to feedback mechanisms.

Declarative Feedback Mechanisms

There are a number of different declarative feedback mechanisms available in swift. One is assertions, which are statements that check for the validity of certain conditions. In debug builds, if an assertion fails, it will cause the program to crash. This can be a useful way to catch errors early on, before they cause more serious problems. Another common declarative feedback mechanism is the use of logging. Logging allows developers to record information about the execution of their code. This information is then used to track down errors, to isolate behaviors in code, and to debug problems.

In addition to assertions and logging, there are a number of other declarative feedback mechanisms available. Error messages provide information about the cause of an error. Warnings are used to indicate potential problems with the code. Of these, here are some specific examples of how declarative feedback mechanisms can be used:

- An assertion that a user-supplied string is not empty before trying to use it.
- Log the values of certain variables at different points in your code.
- Provide a more detailed error message if a user tries to access a nonexistent file.
- Warn a user if they are trying to use a deprecated API.

Of these, the most common feedback mechanism is assertions.

Assertions

As noted, assertions are used to check the validity of certain conditions. There are three types of assertions in Swift:

- assert(::file:line:): This is the simplest type of assertion. It takes a condition and a message as arguments. If the condition is false in debug builds, the assertion will fail and the message will be printed.
- assertionFailure(_:file:line:): This type of assertion is similar to assert(false, ...). In debug builds, if the assertion fails, the message will be printed and the program will crash.
- precondition(::file:line:): This type of assertion is used to check for conditions that must be true before the program can continue. If the precondition fails, the program will crash in debug and optimized builds.

Note Indicating that programs will crash doesn't mean the user will receive an error – it can be done gracefully.

Here is an example of an assertion:

```
assert(1 + 1 == 2, "1 + 1 should equal 2")
```

This assertion will check if the expression 1 + 1 == 2 is true. If it is true, the assertion will pass and the program will continue. If it is false, the assertion will fail and the message "1 + 1 should equal 2" will be printed.

To expand on the use of assertionFailure

```
func divide(_ dividend: Int, by divisor: Int) {
  if divisor == 0 {
    preconditionFailure("Divisor cannot be 0")
  }
  let result = dividend / divisor
  print(result)
}
```

This function takes two integers as arguments, and it divides the first integer by the second integer. If the divisor is equal to 0, the function will call assertionFailure. This will cause the program to crash and print the message "Divisor cannot be 0".

The preconditionFailure function is used to indicate that a certain code path is never expected to be reached, and if it is reached, then the program should crash.

Here is another example of using assertionFailure:

```
func makePerson(name: String, passwordAge: Int) {
  if passwordAge > 30 {
    preconditionFailure("Password age must be less than or equal to 30")
  }
  let person = Person(name: name, passwordAge: passwordAge)
  print(person)
}
```

This function takes two arguments, the name and the passwordAge of a person. If the age is greater than 30, the function will call preconditionFailure. This will cause the program to crash and print the message "Password age must be less than or equal to 30".

To expand on the use of preconditions:

```
func divide(_ dividend: Int, by divisor: Int) {
  precondition(divisor != 0, "Divisor cannot be 0")
  let result = dividend / divisor
  print(result)
}
```

This function takes two integers as arguments, and it divides the first integer by the second integer. The precondition checks if the divisor is not equal to 0. If it is equal to 0, the precondition will fail and the program will crash. The precondition is used to ensure that the function is only called with valid arguments. If the divisor is 0, the function would divide by zero, which would cause an error. The precondition prevents this error from happening. Here is another example of using a precondition – this time for passwordAge like earlier in the section:

```
struct Person {
  let name: String
  let passwordAge: Int
  init(name: String, passwordAge: Int) {
    precondition(passwordAge <= 30, "Password age must be less than or
    equal to 30")
    self.name = name
    self.passwordAge = passwordAge
  }
}
```

This struct represents a person. The initializer for the struct takes two arguments, the name and the passwordAge of the person. The precondition checks if the passwordAge is less than or equal to 30. If it is greater than 30, the precondition will fail and the program will crash (and hopefully do so gracefully, given that's the point of all this.

Conclusion

It might seem like Asynchronous Programming is a slightly more advanced topic than what someone who manages a bunch of devices at a company will need to know. But consider all the scenarios where it can crop up, in addition to Apple embracing these concepts for Declarative Device Management (DDM). For starters, messages for users. Async Await can be used to propagate the information that user might see in the event that they get an error based on their device state. This is the local admins implementation of a similar concept to DDM.

Device admins can take a cue from Apple. If a device falls out of compliance, has a security event streamed to a SIEM, or doesn't check into a device management platform, a set of actions can be prescribed to bring it back into compliance. If the device isn't

opening and closing sockets to web servers constantly but is in sync with them, as we showed how to do with AsyncHTTPClient, webhooks, and polling, then there's no need to wait for an agent that reaches back out to a glorified web server every 15 minutes. Combined with Push Notifications, covered in Chapter 5, this provides a powerful new set of capabilities, potentially as instantly impactful as wiping a device – but with far more granularity (although wiping a device is still an option for those who just prefer the nuke and pave mentality).

CHAPTER 7

Building Quality Software

When used in software development, the term "quality" refers to the degree a software product meets the needs of its users and is free of defects. Quality software is easy to use, maintain, and extend. There are a number of factors that contribute to quality in software development, including

- *Requirements*: The requirements for the software should be clear, complete, and unambiguous.
- *Design*: The software is designed in a way that is easy to understand and maintain.
- *Coding*: The software is written in a way that is free of defects.
- *Testing*: The software is thoroughly tested to ensure that it meets the requirements and is free of defects.
- *Maintenance*: The software must be easy to maintain so that defects can be fixed and new features can be added.

Quality software is important for a number of reasons. First, it can help to improve the user experience. When software is well-designed and easy to use, users are more likely to be satisfied with it. Second, quality software can help to reduce costs. When software is free of defects, it requires less maintenance and support. Third, quality software can help to increase productivity. When software is easy to use, users can be more productive.

C. Edge et al., *Beyond Scripting*, https://doi.org/10.1007/979-8-8688-3035-8_7

There are a number of ways to improve quality in software development. These include

- *Using a quality assurance (QA) process*: A QA process is a systematic way of ensuring that software meets its requirements and is free of defects. For the purposes of this chapter, QA is designed in a few ways – there is regression testing, automated functional testing, and manual QA. Within automated.
- *Using a version control system*: A version control system allows you to track changes to your software and revert to previous versions if necessary.
- *Documenting your software*: Good documentation makes it easier to understand and maintain your software.
- *Using a continuous integration and continuous delivery (CI/CD) pipeline*: A CI/CD pipeline automates the process of building, testing, and deploying your software.

To keep terms clear in this chapter: manual QA means human-led exploratory and scenario testing; automated QA means scripted tests that run repeatedly; functional testing verifies user-visible behavior; regression testing checks that changes do not break previously working behavior; and unit testing validates isolated functions/classes.

Notice that all of this contributes to building software that is consistent. The software is designed, built, maintained, and tested in a consistent manner. Chapter 1 covered CI/CD, and Chapter 2 explored getting in front of errors with unit testing. So this chapter will start with version control, get into various forms of testing, and then end with some common issues many encounter with secure coding techniques. First, though, let's cover writing efficient code.

Efficient Code

Efficiency isn't just about speed; it's about code that is clear, concise, and performs optimally. Efficient code not only simplifies maintenance and debugging but also enhances the overall performance and stability of your applications.

Before venturing into optimization, think about standardization. Solidify understanding of core programming concepts like algorithms, data structures, and memory management. Don't reinvent the wheel. Use language-specific libraries and frameworks that offer optimized functions and data structures for common tasks. Eschew obfuscated code in favor of transparent and well-structured logic. Clear code is easier to understand, maintain, and debug. Give variables and functions descriptive names that accurately reflect their purpose. This clarifies intent and reduces cognitive load for future readers. Also establish and adhere to consistent indentation, spacing, and naming conventions. This enhances code readability and maintainability.

Aside from standardization, also consider optimization. Analyze your problem and select the most efficient algorithm for the task at hand. Consider factors like time complexity and space complexity. Cache previously computed results to avoid redundant calculations, especially for recursive functions. This is commonly referred to as memoization. Also check for exit conditions early in your loops and conditional statements to avoid unnecessary iterations.

As code grows, also consider memory management. Avoid unnecessary memory allocations by reusing existing objects and data structures where possible. Close files, release resources, and handle object lifecycles effectively to prevent memory leaks. Utilize profiling tools to identify memory bottlenecks and optimize memory usage within your code.

Beyond memory management, also consider performance profiling. Don't optimize blindly. Employ profiling tools to identify performance bottlenecks and understand where your code spends its time. Focus optimization efforts on the areas identified by profiling. Also, regularly benchmark code to track performance changes and measure the impact of optimization efforts.

It's also important to keep code modular, which is covered later in the chapter when discussing cyclomatic complexity. But make sure to break down tasks, which means divide code into smaller, reusable functions and modules. This improves maintainability, reusability, and testability. Abstraction techniques will hide complex implementation details and provide clean interfaces for other parts of the code. Aim for loosely coupled components that depend on each other minimally. This enhances modularity and simplifies maintenance.

Finally, don't fear refactoring. Continuous improvement means that code evolves with project progression. Break down large refactoring tasks into smaller, incremental steps to minimize risk and make the process manageable. There are tons of automated

tools that can help make all of these changes easier. For example, utilize code formatters and static analysis tools to help automate basic refactoring tasks and maintain code quality. There are plenty of code formatters on the Xcode Extension store (Figure 7-1), available under the "Xcode" menu of Xcode.

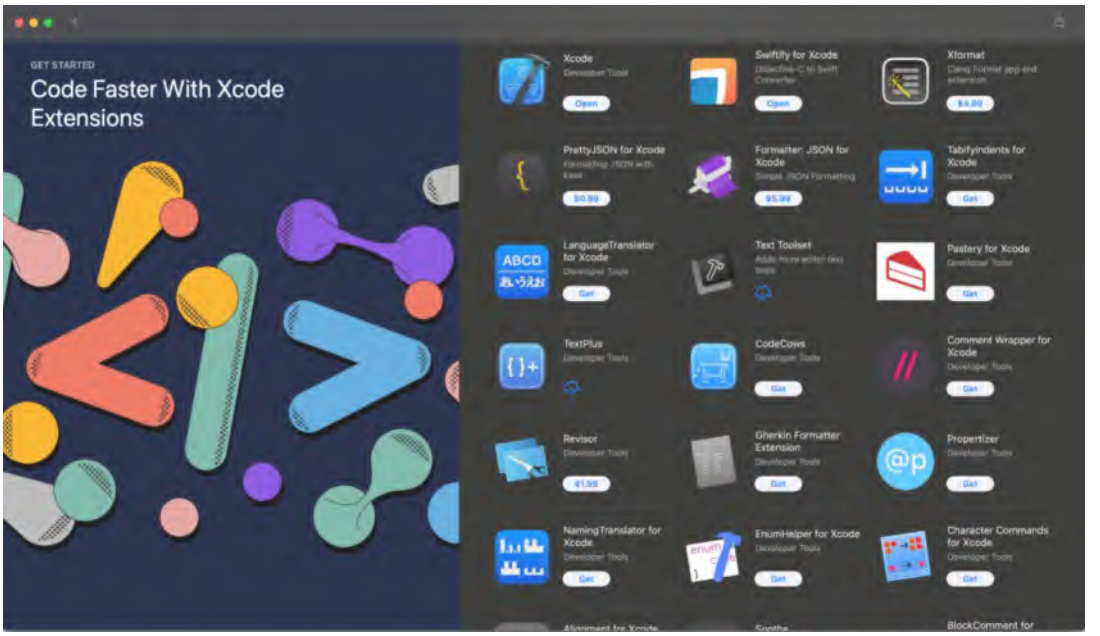

Figure 7-1. *Xcode extensions store*

As those refactoring tasks proceed, if it hasn't already happened, it's a good idea to go ahead and start version controlling Xcode projects.

Before You Software, Version Control

Software version control is a system that helps software developers track and manage changes to their code. It allows developers to work on different versions of the code at the same time and to easily revert to previous versions if necessary. There are many different software version control systems available, such as Git (with GitHub and GitLab), Mercurial, and Subversion (one of the oldest, written in 2000). Each system has its own strengths and weaknesses, so it's important to choose one that's right for your needs. Version control is not just backup; it is also a quality tool because it enables review, rollback, branching, traceability, and safer experimentation.

Once a software version control system is selected, choose a strategy for using it (although in the beginning, if there's not much experience with them, that might just be "everything goes to the repo"). Here are a few things to keep in mind:

- Commit early and often. This will help track changes and make it easier to revert to a previous version if necessary.
- Use descriptive commit messages. This will help the team understand what changes were made in each commit. Even if there's just a team of one for now, that will help a future team or replacement to understand what, why, and how various options were implemented in code.
- Use branches. Branches allow developers to work on different versions of the code at the same time. This is a great way to experiment with new features or fix bugs without affecting the main branch of the code.
- Merge changes regularly. This will help keep code up-to-date and avoid conflicts.
- Use a continuous integration (CI) server. A CI server can automatically build and test code after each commit. This is a great way to catch errors early on and ensure that code is always working. It also makes it easier to go get a cup of coffee or eat, for those who do such things.
- Involve the team. The best software version control strategies are the ones that are created by and for the team that will be using them. Get input from team members on what features are important to them and how they would like to use the system, if there is a team. If not, there's probably a development team somewhere in the organization it's good to cross-train them on what's going on as well.
- Start small. Don't try to implement a complex software version control strategy all at once. Start with a simple strategy and then add more features as needed.

- Be flexible. Software development is an iterative process, so software version control strategies should be flexible enough to adapt to changes as well. Be prepared to make changes to the strategy as needs evolve.
- Use the right tools. There are many different software version control systems available. Choose the one that's right for the specific needs of a given project.
- Get training. If not familiar with software version control, get training from a qualified instructor. This will help learn how to use the system effectively and avoid common mistakes.
- Back up the code. It's not unheard of to wipe out an entire repo during rebasing, etc.

One other thought is that when trying to manage everything else the way software is managed, it often relies on custom scripts that wrap up API endpoints and custom workflows. Consider breakpoints at each step along the way to get keys where they need to go, etc. Given that the two most common tools used to interface with version control for Swift environments are GitHub as the version control system and Xcode as the IDE, let's look at how to set up Xcode to store projects on GitHub.

GitHub and Xcode

For the purposes of this section, developers will need an account with GitHub. Working with other git tools like GitLab is similar as well. Further, Xcode 12 or greater will be required, as will be a GitHub personal access token for secure authentication from Xcode. Armed with those, open Xcode, click the Xcode menu, then click Settings, and select Accounts. At the Accounts screen, click the plus sign to add a new account (unless of course there's already an account).

At the modal, click GitHub (unless access to another service is being configured) and click Continue, as seen in Figure 7-2.

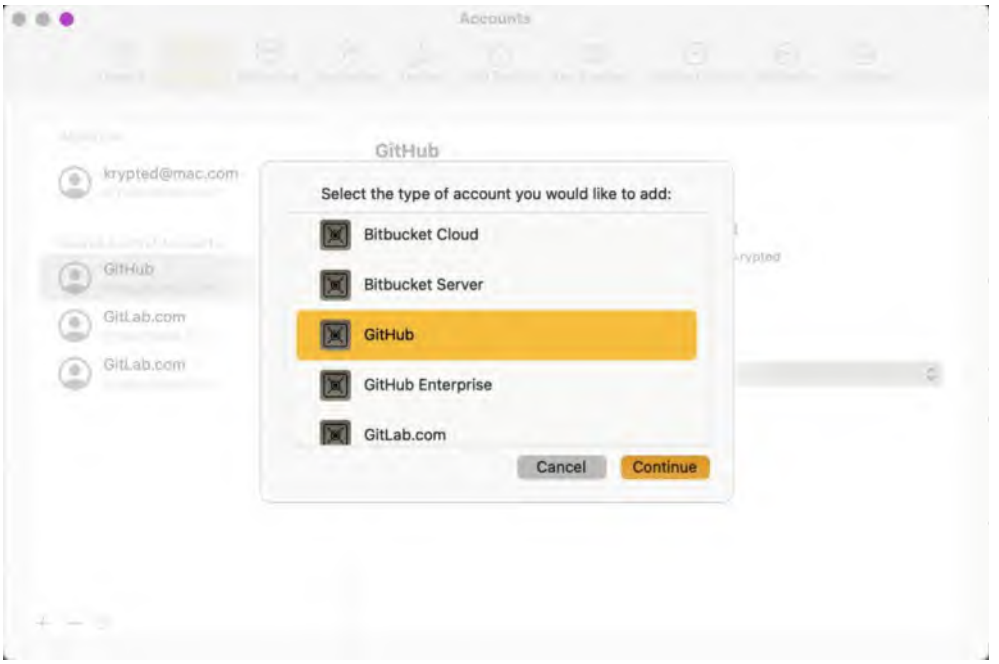

Figure 7-2. *Setting up a GitHub account in Xcode settings*

At the sign-in screen, enter the appropriate GitHub credentials and authorize Xcode (Figure 7-3). If there aren't any yet, click the Create a Token on GitHub button.

Figure 7-3. *Configure GitHub authentication in Xcode settings*

If creating a new token, it's usually a good idea to have one per user or device, as seen in Figure 7-4. The button will open the link to GitHub's token page, where admins can scroll down to the bottom of the page and click Generate Token once all permissions the token will have been configured as desired.

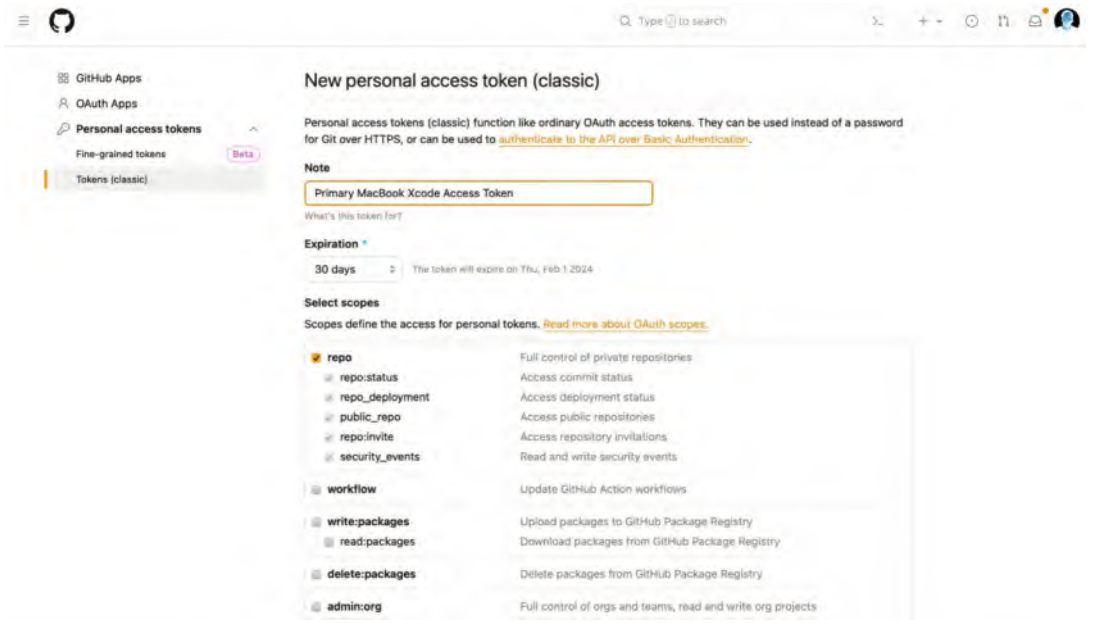

Figure 7-4. *Creating an access token in GitHub*

Once the token is provided, click Sign In, and once the GitHub connection is complete, the page will reflect and ask to update passwords every now and then, as seen in Figure 7-5.

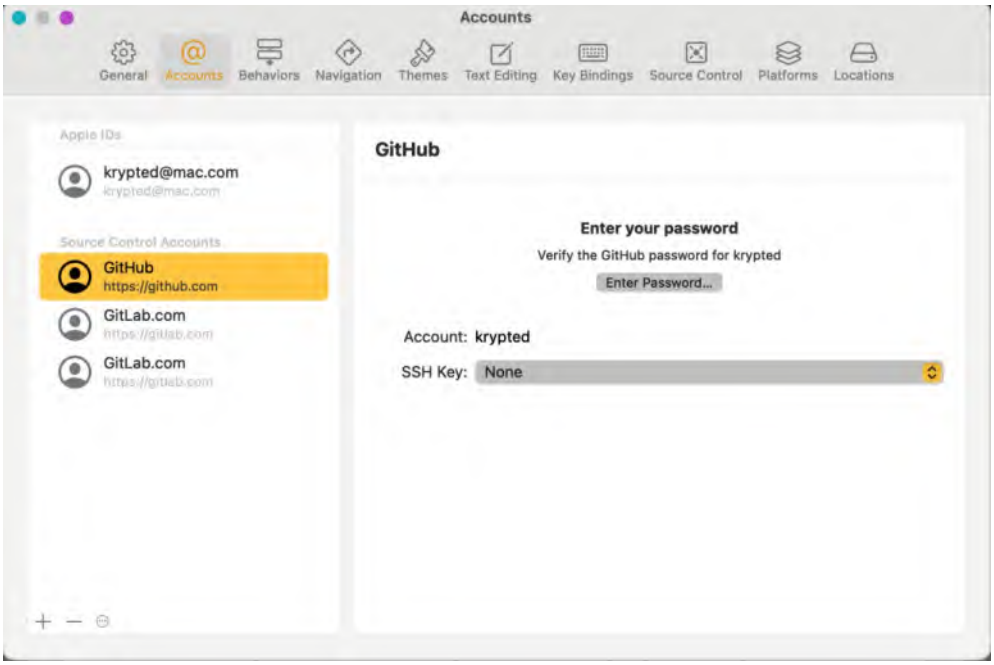

Figure 7-5. *GitHub account in Xcode settings*

Now that the account is linked to GitHub, it can be used to create new projects, provided that those permissions were provided to the Access Token. To do so, from Xcode, choose File, then New, and Project. Provide a name and location for the local instance of the new project, and then make sure to enable Git using the "Create Git repository on my Mac" checkbox as seen in Figure 7-6.

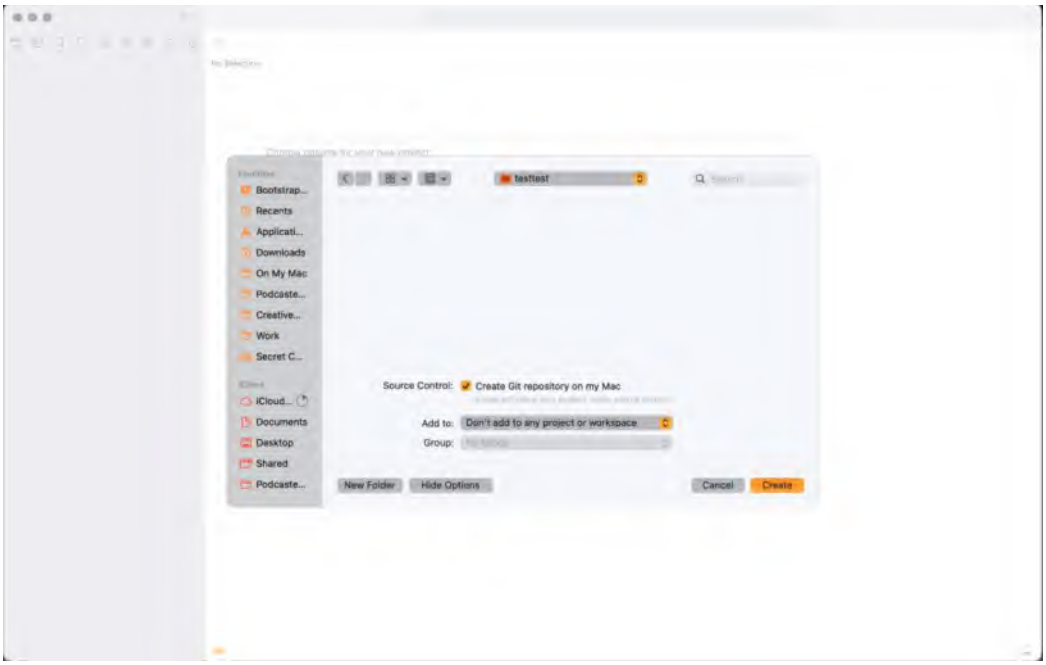

Figure 7-6. *Creating a git repository*

Once the project is created, it's time to push it to GitHub. Click View, then Navigators, then Source Control to open the Source Control navigator. Right-click the "Remotes" folder and select "New Remote." Enter the URL of the GitHub repository and click "Create." Commit your initial changes and push them to GitHub using the "Push" button.

To work with an existing GitHub repository, clone the repository to the local host. To do so, first copy the repository's SSH or HTTPS URL from GitHub. To access that, open the repository on GitHub and click on the Code button, which shows the URL in the flydown menu, as seen in Figure 7-7.

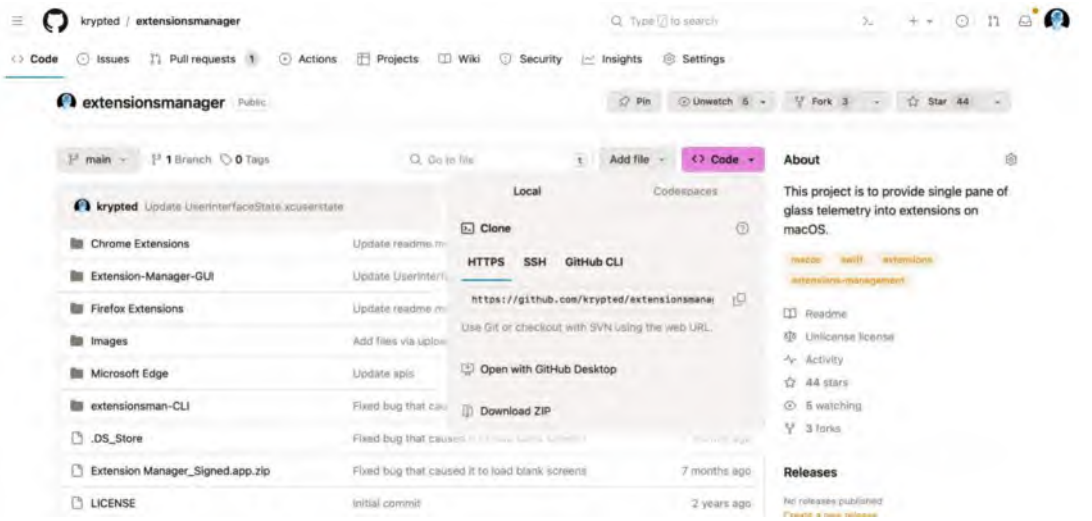

Figure 7-7. *Cloning an existing GitHub repository*

Armed with the URL, open Xcode, select the Window menu, and click on "Welcome to Xcode" opting for the Clone Git Repository… menu item as seen in Figure 7-8.

Figure 7-8. *Welcome to Xcode Window*

Paste the URL and click “Clone.”

It’s now possible to use version control to track changes, revert to previous versions, and manage code history effectively. Use the Source Control navigator to browse changes that haven’t been committed, see branches, etc., as seen in Figure 7-9.

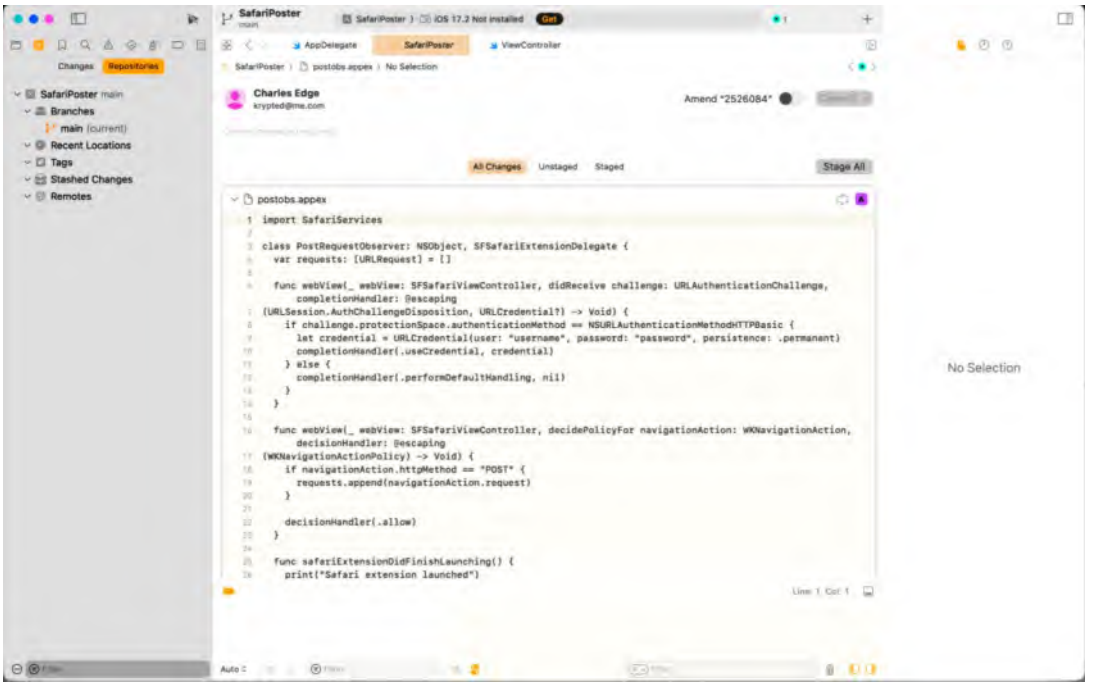

Figure 7-9. *Source Control Navigator in Xcode*

Version control allows admins to work seamlessly with team members, merging code and resolving conflicts efficiently. It also allows for branching and merging and approving pull requests, which is how others can review and merge code contributions in a structured manner. However, there are plenty of challenges that this all creates. These can make the git commands that the graphical interface runs on behalf of developers a bit less than what might be desired. Therefore, many will invariably need to turn to the git command line interface.

Use git from the Command Line

Git is easy. It's a command with some verbs. Let's start with the init verb, which creates an empty git repository in the working directory (or supply a path after the verb):

```
git init
```

Now let's touch a file in that directory. Once a new file is there, with that new repo as your working directory, run git with the status verb:

```
git status
```

Oh look, now it shows the branch:

```
On branch master
```

There's also a message, provided there haven't been any commits, that indicates such:

```
No commits yet
```

If there are any untracked files, run git with the add verb, specifying a file or path (or use . when the working directory is still the directory of the path).

```
git add .
```

Now let's run that status command again. And hey, it sees that there is staged file (or files) if there are. This takes tracked and staged files and commits them. Until this step, it's easy to revert back to the previous state of that file (which in this simple little walkthrough would be for the file to no longer exist).

```
git commit -m "test"
```

Now let's edit the file and then once done, run git with the diff verb:

```
git diff
```

This allows admins to change between file(s). Check out the logs to see what you've been doing to poor git:

```
git log
```

The logs list commits, along with an author, a date and time stamp, as well as a name of the file(s) in the commit. Now, let's run a reset to revert to our last commit. This will overwrite the changes just made prior to doing the diff (use a specific commit by using it as the next position after --hard or just leave it for the latest actual commit):

```
git reset --hard
```

This resets all files back to the way it was before mucking around with those poor files. Before exploring the wide world that is cloning a repository off the interwebs, first do a quick look at branches given that most shouldn't work in a master branch for a number of reasons. So let's look at existing branches by running git with the branch verb:

```
git branch
```

Note that there's a branch, the "main" branch. To create a new branch, simply type git followed by the name of the branch to create (in this case, it will be called myspiffychanges1):

```
git branch myspiffychanges1
```

Run git with the branch verb again to see that below master, the new branch appears. The asterisk is always used, so keep track of the active branch. To switch between branches, use the checkout verb along with the name of the branch:

```
git checkout myspiffychanges1
```

This could have been done in both of the previous steps in one command, by using the -b flag with the checkout verb, as follows:

```
git checkout -b myspiffychanges1
```

Now the asterisk should be on the new branch to enable making changes there. Let's edit that file from earlier. Then let's run another `git status` and note that the modifications can be seen. Then, let's add them to the list of tracked changes using the `git add` – again for the working directory:

```
git add .
```

Now let's commit those changes:

```
git commit -m "some changes"
```

And now there are two branches, a little different from one another. Let's merge the changes into the master branch next. First, let's switch back to the master branch:

```
git checkout master
```

And then let's merge those changes:

```
git merge myspiffychanges1
```

OK – so now it's possible to init a project, branch, and merge. Before going onto the internet, let's first set up a name. Notice in the logs that the Author field displays a name and an email address. Let's see where that comes from:

```
git config --list
```

This is initially populated by ~/.gitconfig, so let's edit that next. Or let's remove what is in that list:

```
git config --global --unset-all user.name
```

And then add a new set of information to the key to edit:

```
git config user.name "Charles Edge" --global
```

Let's configure an email address too, so people can yell at the right person for their crappy code some day:

```
git config user.email "chuckufarley@me.com" --global
```

OK, optics aside, let's clone an existing repository onto the local computer. The clone verb allows developers to – insert suspense here – clone a repository into the home directory, like doing so with autopkg here:

```
git clone https://github.com/autopkg/autopkg
```

The remote verb makes a local copy of a branch. But it takes a couple of steps. First, init a project with the appropriate name and then cd into it. Then grab the URL from GitHub (Figure 7-10):

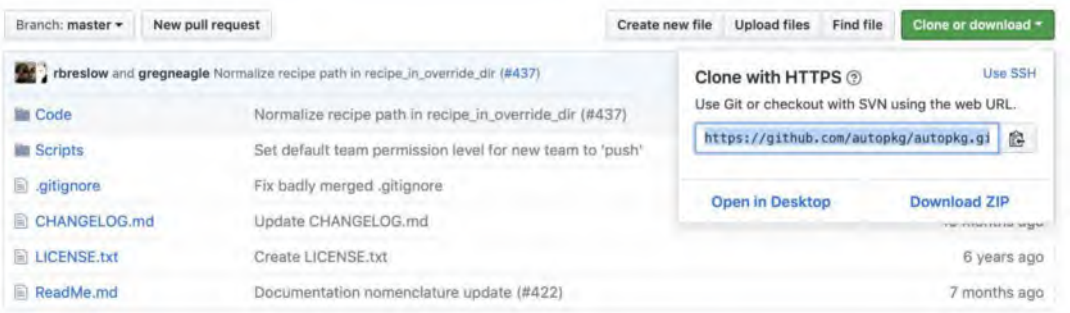

Figure 7-10. *Using HTTPS to clone a repository in GitHub*

And add it using the remote verb:

```
git remote add AutoPkg https://github.com/autopkg/autopkg.git
```

Now let's fetch a branch of that project, in this case, master:

```
git fetch AutoPkg master
```

Now download the contents of that branch:

```
git pull AutoPkg master
```

And once there are some changes, let's push our changes:

```
git push AutoPkg myspiffychanges1
```

Other common tasks include rebasing, forking, pulling, and plenty of other aspects. But these will get any admin started with using git and version control in general. Now that code can be checked in and changes followed, and even approved, by other admins, let's get into quality assurance, or QA.

Manual QA

The most approachable of software testing options is manual QA. Manual QA is the process of testing software, as the name implies, manually. Automated QA automates this same process, either with unit tests or with automated functional tests. Manual QA is done by a human tester who interacts with the software and looks for defects. Manual QA can be more flexible than automated QA. For example, manual QA can be

used to test edge cases or unusual scenarios that might not be caught by automated tests. Manual QA can also be more accurate than automated QA. This is because human testers can use their judgment to identify defects that automated tests might miss.

Manual QA results can be more easily interpreted than automated QA results. This is because human testers can explain why they think a defect exists. There are a number of reasons for this. One is that unit tests check code to see if it reflects a given desired state. That state might not be reflected in the graphical interface, because most large software projects (especially in a tightly object oriented project) sprawl and when a butterfly flaps its wings in one function, even if just a little bit, it can have resounding effects elsewhere. The same type of thing can be true in automated functional testing, which usually matches graphical patterns – a small change in an operating system can then cause all automated functional QA tests to fail.

There are a number of ways to do manual QA, which often include

- *Exploratory testing*: Exploratory testing is a type of manual testing where the tester does not have a specific plan in mind. Instead, the tester explores the app and looks for defects.
- *Boundary testing*: Boundary testing is a type of manual testing where the tester tests the limits of the app. This includes testing for unexpected inputs, invalid inputs, and out-of-range inputs.
- *Usability testing*: Usability testing is a type of manual testing where the tester tests the usability of the app. This includes testing how easy it is to use the app, how intuitive the app is, and how well the app meets the needs of the users.

The type of testing done for apps is usually determined by what the app does. It's more important to be organized than to think too much about the labels applied to the types of testing – or at least until a team gets large enough that new developers scoff at the attempts of the original team to flounder about with what they think of as an entire discipline. But the earlier and more documented the attempts the more historical and/or institutional knowledge the newer people will have.

There are a number of tools that can be used for manual QA for Swift apps. These include

- *Xcode*: Xcode is Apple's integrated development environment (IDE). It includes a number of tools that can be used for manual QA, such as the debugger and the console.

- *TestFlight*: TestFlight is Apple's beta testing platform. It can be used to distribute beta versions of an app to testers so that they can manually test it.
- *Slack*: Slack is a messaging platform that can be used to communicate with testers. This can be helpful for coordinating manual testing activities and sharing feedback.

But the most important tool to keep organized is probably what we'll just call the "QA Matrix" for the purposes of this chapter.

The QA Matrix

A QA matrix is a tool that can be used to help ensure the quality of software. It is a table that lists the different aspects of the software that need to be tested, the different types of tests that can be performed, and the severity of the defects that could be found.

To make a QA matrix for software developers, first

- Identify the different aspects of the software that need to be tested. This could include the user interface, the functionality of the software, the performance of the software, and the security of the software.
- Identify the different types of tests that can be performed. This could include unit testing, integration testing, system testing, and user acceptance testing.
- Assign a severity level to each defect. This could be a scale of 1 to 5, with 1 being the least severe and 5 being the most severe.
- Create a table that lists the different aspects of the software, the different types of tests that can be performed, and the severity of the defects that could be found.

Table 7-1 shows an example of a QA matrix for software developers.

Table 7-1. *Sample QA Matrix for software developers*

Aspect of Software	Type of Test	Severity
User interface	Unit testing	1
Functionality	Integration testing	2
Performance	System testing	3
Security	User acceptance testing	4

The QA matrix can be used to help ensure that the software is tested thoroughly. It can also be used to prioritize the testing efforts. For example, if there are a number of high-severity defects, those defects should be prioritized for testing. The QA matrix can be a valuable tool for software developers.

Here are some additional tips for creating a QA matrix:

- Be as specific as possible when identifying the different aspects of the software that need to be tested. This will help to ensure that the testing is thorough.
- Include all of the different types of tests that can be performed. This will help to ensure that all of the potential defects are found.
- Assign the correct severity level to each defect. This will help to prioritize the testing efforts.
- Keep the QA matrix up-to-date. As the software changes, the QA matrix should be updated to reflect the changes.

Once what should be in the matrix is understood, it's time to build a process where it gets used every time a build is done. In smaller teams, this might be the developer taking it through its paces. In larger teams, there might be someone in support that helps out, some people in a larger open source community who do testing, or within a company, people in support or a dedicated QA team.

The manual QA work then also sets the stage for automated functional testing.

Automated Functional QA

This is where Selenium, the open source web automation framework, steps in, offering a surprising twist: performing automated functional testing of Swift apps. Appium is a popular mobile automation framework that extends Selenium's capabilities to native, web, and hybrid iOS and Android apps. It seamlessly integrates with XCTest, iOS's native testing framework, for robust test execution.

To install Appium, first make sure the dependencies are installed. Check `java --version` and for a recent copy of Homebrew (using the brew command), as seen in Figure 7-11.

```
charlesedge@CE-MacBook-Pro ~ % java --version
java 16 2021-03-16
Java(TM) SE Runtime Environment (build 16+36-2231)
Java HotSpot(TM) 64-Bit Server VM (build 16+36-2231, mixed mode, sharing)
charlesedge@CE-MacBook-Pro ~ % brew -v
Homebrew 4.0.4
Homebrew/homebrew-core (git revision 52d0e2450d1; last commit 2023-03-01)
Homebrew/homebrew-cask (git revision 6e832f93b1; last commit 2023-03-01)
```

Figure 7-11. *Checking Java version from the command line*

Install Appium and Dependencies: Set up Appium on your preferred platform (desktop or cloud) and ensure you have the necessary libraries (WebDriver, XCTest, Appium-Swift bindings) installed. Next, run the node command to see if there's a recent version of node installed. If the command fails, then node likely isn't installed, but can be installed quickly using Homebrew:

```
brew install node
```

Next, we'll need some small UI automation scripts available via npm and Homebrew:

```
npm install -g authorize-ios
brew install ios-deploy
```

Note The npm command might not be necessary according to the versions being run, so if there's an error that it was moved into Appium, just disregard and move on.

Now to install Appium using npm:

```
npm i --location=global appium
```

Next, make sure Appium runs by running the `appium` command from the command line:

```
appium
```

The result should appear similar to the following (note the port number that Appium is bound to):

```
[Appium] Welcome to Appium v2.3.0
[Appium] The autodetected Appium home path: /Users/charlesedge/.appium
[Appium] Appium REST http interface listener started on http://0.0.0.0:4723
```

Finally, it's time to write scripts in the preferred programming language. If that is Python, run the following to install the Appium Python client:

```
pip install Appium-Python-Client
```

Appium can then interact with Swift app elements, including through Python scripts. To locate UI elements, use accessibility IDs, XPath, or iOS-specific methods like using element name or predicate. To initiate the Appium session, specify the desired iOS device or simulator and the app's bundle identifier.

It's then possible to use Appium commands to perform actions on UI elements, simulating user interactions like tapping buttons, entering text, or swiping menus. For example, the following script imports the necessary Appium and Selenium libraries for element locators and then sets the necessary capabilities and defines the platform, device, automation name, and app path. The script then connects to the Appium server and starts a session, waits for UI elements to become available (for up to 10 seconds), and handles the potential timeout if the element isn't found. It then closes the Appium session:

```
from appium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException
# Desired capabilities for the Appium session
```

```
desired_caps = {
   "platformName": "iOS",  # Replace with "Android" if testing on Android
   "deviceName": "iPhone 14",  # Replace with your device name or
   simulator name
   "automationName": "XCUITest",  # Use XCUITest for iOS apps
   "app": "/path/to/com.krypted.tester.app"  # Replace with the path to
   your app
}
# Initialize the Appium driver
driver = webdriver.Remote("http://localhost:4723", desired_caps)
# Wait for the app to launch and the UI element to be available
try:
   element = WebDriverWait(driver, 10).until(
       EC.presence_of_element_located((By.NAME, "testUIelement"))
   )
   print("UI element 'testUIelement' is present.")
except TimeoutException:
   print("UI element 'testUIelement' not found.")
# Close the Appium session
driver.quit()
```

There are simpler and far more complicated tests that can be run. But the power of Python (or JavaScript or whatever language is preferred) combined with the power of a GUI framework and the libraries installed previously allows for complex GUI automations and tests. Appium is based on Selenium, so supports assertion capabilities to verify the expected behavior of an app after each action. For example, checking if a text label displays the correct value after tapping a button. It also supports mechanisms to generate comprehensive test reports and logs, helping analyze successes and failures.

Finally, consider refactoring test scripts to utilize the Page Object Model (POM) design pattern, promoting code reusability and maintainability. Also leverage data files to drive different test scenarios with various parameter values, increasing test coverage and efficiency. It's also possible to run multiple tests concurrently on different devices or simulators to reduce overall test execution time – just be careful to consider that there is parallelization when doing so.

Automated QA

Automated functional QA can save on software testers. But it often requires far more expensive resources to automate tasks that people on a helpdesk might be able to do – and so is usually done in larger teams. Many think of automated QA as automated tests to be run when compiling software. This can help to improve the quality of software by catching defects early in the development process and was covered in further detail in Chapter 2.

There are a number of ways to do automated QA for Swift apps. These include

- *Unit testing*: Unit testing is a type of automated testing that tests individual units of code.
- *Integration testing*: Integration testing is a type of automated testing that tests how different units of code interact with each other.
- *System testing*: System testing is a type of automated testing that tests the entire system.
- *UI testing*: UI testing is a type of automated testing that tests the user interface.

There are a number of tools that can be used for automated QA for Swift apps. These include XCTest, Apple's built-in unit testing framework covered in Chapter 2. XCUITest is Apple's built-in UI testing framework. There are also third-party frameworks, like Quick, Specta, and KIF.

Unit Testing

Unit testing is a way of testing code in isolation. This means that single units of code are tested, such as a function or a class, without having to worry about how it interacts with other parts of the code. Doing so improves code quality; unit tests can help identify and fix bugs in code. Unit testing also makes code easier to maintain by helping to make changes easier and quicker as there's less concern about a change in one part of code negatively impact other areas of code. In short, unit testing makes code less brittle.

To write a unit test in Swift, create a XCTestCase subclass. This subclass will contain the methods to test. Here is an example of a XCTestCase subclass:

```
class MyTests: XCTestCase {
  func testAdd() {
    let expected = 1 + 1
    let actual = MyClass.add(1, 1)
    XCTAssertEqual(expected, actual)
  }
}
```

This code creates a MyTests subclass that contains a single test method called testAdd(). The testAdd() method tests the add() method of the MyClass class. The XCTAssertEqual() method is used to compare the expected and actual results of the test. If the expected and actual results are not equal, then the test will fail.

To run unit tests in Xcode, use the Product menu and then select the Test command. This will run all of the unit tests in a given project and report back any failures. However, different variations can cause different behaviors – thus regression testing.

Regressions

A regression is a defect that causes software to behave differently after a change has been made to it. This can be caused by a number of factors, such as a bug in the code, a change in the environment, or a compatibility issue. Regressions can be difficult to find and fix, as they may only occur under certain conditions. This can make it difficult to reproduce the defect, which can make it difficult to debug.

There are a number of ways to prevent regressions. Using a version control system: A version control system allows tracking changes to code and reverting to previous versions if necessary.

Testing thoroughly can help to identify and fix regressions before they are released to production. As software grows, it may become necessary to use a regression testing suite. There are a number of tools, in addition to those provided by Apple that go deeper into specific regression analysis. These include

- *EarlGrey*: Developed by Google for testing their iOS apps, now open source. Emphasizes synchronized interactions with UI elements, ensuring test stability and reliability. Offers a variety of powerful assertions for verifying UI behavior and state. Allows for custom matchers and extensions to tailor testing to specific needs.

- *Quick and Nimble*: Embraces Behavior-Driven Development (BDD) principles for writing clear and concise test cases that focus on expected behavior. Provides a framework for defining test cases in a human-readable format and a rich set of matchers for making assertions about results. It can also be used with XCTest.
- *Calabash*: Also promotes BDD, but using the Cucumber language for writing test cases in a human-readable format. This makes tests cross-platform with Android.
- *TestProject*: Cloud-based platform for test automation that supports iOS, Android, web, and API testing with a visual interface and a large community of testers, making it amongst the most approachable.

Keep in mind that there are two main types of regressions: functional regressions and performance regressions. Functional regressions are defects that cause the software to behave differently than it was designed to behave. Performance regressions are defects that cause the software to run slower or use more resources than it was designed to. Going beyond straight regression testing is fuzzing. Here, random bits of data are placed into a number of places to determine if doing so can cause a program to crash.

Fuzzing

Sometimes it's important to test a function to see how robust it is. This is a small example fuzzing function to input randomly generated bytes that get passed to another function. It just uses random bytes so much more logic could easily be added to constrain what's being passed to whatever type it's being passed to, but this satisfies many a need to get started.

```
import Foundation
func fuzz(function: (Data) throws -> Void) {
  // Generate a random input to pass to the function we are fuzzing
  let input = Data((0..<1024).map { _ in UInt8.random(in: 0...255) })
  // Call the fuzzed function with the random input
  do {
    try function(input)
  } catch {
```

```
    print(error)
  }
}
```

There's no consistent way to write or run unit tests. It's really more a matter of knowing the code and looking for ways to break it, in order to make it more resilient and secure.

Reducing Cyclomatic Complexity

Cyclomatic complexity is a metric used to measure the complexity of a software module. It is calculated by counting the number of linearly independent paths through the code. A path is a sequence of statements that can be executed from the beginning of the module to the end. A path is considered to be linearly independent if it cannot be combined with any other path to form a longer path.

The cyclomatic complexity of a module is calculated by counting the number of decision points in the module. A decision point is a statement in the module that can cause the control flow to branch to different paths.

For example, the following code has a cyclomatic complexity of 3:

```
if x == 1 {
  y = 1
} else if x == 2 {
  y = 2
} else {
  y = 3
}
```

There are two decision points in the above code: the if statement and the else if statement. Cyclomatic complexity is the number of decision points plus one, so the cyclomatic complexity of the code is 3. This way of thinking about cyclomatic complexity can be a good indicator of the maintainability and testability of code. High cyclomatic complexity makes code more difficult to understand, maintain, and test.

There are several ways to keep cyclomatic complexity to a minimum. Try to use simple control flow statements instead - like if and for, whenever possible. Avoid using more complex control flow statements, such as switch and case, unless necessary. Also

consider breaking down any long functions and methods into smaller, more manageable stanzas of code. Use conditional logic sparingly. When using conditional logic, make sure that the conditions are clear and easy to understand. Use comments to explain the purpose of code to keep it readable and easier to understand, although comments do not reduce cyclomatic complexity.

It's never too early to use a linter. A linter is a tool that can help you to identify potential problems in code, including cyclomatic complexity. There are a number of linters available for Swift, such as SwiftLint: `https://github.com/realm/SwiftLint`. The security implications of writing code that repeats itself or has overly branching logic is that it's hard to propagate changes throughout the codebase when security or other issues are found. One of those types of things that crops up is memory safety.

Memory Safety

Memory safety is an important aspect to understand about any programming language. Memory safety features prevent errors like buffer overflows and dangling pointers, which can lead to crashes, security vulnerabilities, and other problems. Memory leaks can still occur. Apple has introduced a number of memory safety features, which (at least theoretically) include the following:

- *Automatic reference counting (ARC)*: ARC is a memory management system that automatically tracks the ownership of objects and deallocates them when they are no longer needed. This eliminates the need for developers to manually manage memory, which is a common source of errors.
- *Bounds checking*: Swift performs bounds checking on all array and string accesses to ensure that indices are within range. This prevents buffer overflows, which can occur when code writes to memory outside of the bounds of an array or string.
- *Type safety*: Swift's type system helps to prevent memory safety errors by ensuring that values of the correct type are always assigned to variables and used as arguments to functions. This helps to prevent errors such as passing a pointer to the wrong type of object.

- *Exclusive access*: Swift requires that code modifying a location in memory have exclusive access to that memory. This prevents overlapping access violations.
- *Warnings*: The Swift compiler can issue warnings about potential memory safety errors. Developers can enable these warnings in their project settings to help catch errors early.
- *Static analysis*: Swift provides a number of static analysis tools that can check code for potential memory safety errors. These tools can be used to analyze code before it is compiled, which can help to prevent errors from being introduced into the codebase.

Some of these concepts will make sense as they're explained further in the following sections, starting with access control.

Access Control

Access control allows developers to control who can access and modify code. This can be useful for hiding implementation details, protecting sensitive data, and preventing errors. There are six access control levels, which include the following:

- *Open*: Open classes and class members can be accessed and subclassed from anywhere.
- *Public*: Public classes and class members can be accessed from anywhere, but they cannot be subclassed outside of their defining module.
- *Package*: Package classes and class members can be accessed from within the same Swift package.
- *Internal*: Internal classes and class members can only be accessed from within the same module.
- *Fileprivate*: Fileprivate classes and class members can only be accessed from within the same file.
- *Private*: Private classes and class members can only be accessed from within the same class or extension.

To declare the access level for a class, struct, enum, property, method, initializer, or subscript, use one of the following access control modifiers before the declaration:

```
open class MyClass {}
public struct MyStruct {}
package enum MyEnum {}
fileprivate func myFunction() {}
private var myProperty = 0
```

By default, classes and class members are internal, and properties and methods are also internal unless otherwise constrained. This means developers need to explicitly declare the access level for any class, struct, enum, property, method, initializer, or subscript that should be accessible from outside of its defining module or class. A class or class member can only be accessed from code that has the same access level or a higher access level. Further, a property can only be modified from code that has the same access level or a higher access level. A method can only be called from code that has the same access level or a higher access level.

When using access control, only expose the public interface of code. Then use access control to hide implementation details, protect sensitive data, and prevent errors. In addition to classes, properties, and methods, access control can be set in modules and files.

Modules and Files

Classes and class members are internal, by default. This means that they can only be accessed from within the same module. The access control levels are the same as above when defined explicitly. To make a class or class member accessible from outside of its defining module, declare it as public or open.

For example, the following class is declared as public:

```
public class MyClass {}
```

This means that MyClass can be accessed from outside of its defining module. Properties work similarly, so to make a property or method accessible from outside of its defining class or extension, declare it as public or internal.

For example, the following property is declared as public:

```
public var myProperty = 0
```

This means that myProperty can be accessed from anywhere in your code, regardless of which file it is defined in, including in multi-target apps.

Access Control in Multi-target Apps

Multi-target apps allow developers to create a single codebase that can be compiled for multiple platforms, such as iOS, macOS, and watchOS. However, multi-target apps also make access control more complex because developers need to consider how access control will work across different platforms.

For example, exposing a public class for iOS might need to be kept private on macOS. Or exposing a public method on watchOS that stays internal on iOS. To achieve this, use access control modifiers to explicitly declare the access level of classes, structs, enums, properties, methods, initializers, and subscripts. This will ensure that code is accessible from the correct platforms. Also consider conditional access control based on the platform that it is compiled for. This can be useful for hiding implementation details or protecting sensitive data. Frameworks will then help to modularize code and encapsulate implementation details. This can make it easier to control access to code from different platforms.

Let's look at how some of these look when used in code. For example, to declare a public class on iOS, but keep it private on macOS, os(iOS) can be used in an if statement:

```
#if os(iOS)
public class MyClass {}
#else
private class MyClass {}
#endif
```

To use conditional access control to control access to a method rather than a class based on the platform that it is compiled for, the following can do the same with watchOS as the os()::

```
#if os(watchOS)
public func myMethod() {}
#else
internal func myMethod() {}
#endif
```

To use a framework to modularize code and encapsulate implementation details, first create the framework:

```
public struct MyFramework {
    // ...
}
```

To then expose the public interface of a framework from different platforms, a similar process is followed:

```
#if os(iOS)
import MyFramework
```

Then to access the public interface of MyFramework on iOS conditionally:

```
let myFramework = MyFramework()
#endif
#if os(macOS)
import MyFramework
```

Or for macOS:

```
let myFramework = MyFramework()
#endif
```

Keep in mind, once some aspects are explicitly defined, as projects grow, developers may need to explicitly declare the access level of all classes, structs, enums, properties, methods, initializers, and subscripts. Also, new operating systems come along, like iPadOS or visionOS, so consider the future when choosing whether to nest access control inside an if or an else.

Access Control and Variables and Constants

Finally, it's also possible to limit (or broaden) the access control level for a variable or constant. To do so, use one of the following access control modifiers before the declaration (which match the six levels defined earlier):

```
open class MyOpenClass {
open var myVariable = 0
}
```

```
public let myConstant = 1
package var myPackageVariable = 0
internal var myInternalVariable = 2
fileprivate let myFileprivateConstant = 3
private var myPrivateVariable = 4
```

By default, variables and constants are internal. This means that they can be accessed from within the same module. To make a variable or constant accessible from outside of its defining class or extension, you must explicitly declare it as public or internal. To expose a public variable to allow other code to read its value, but prevent other code from modifying it:

```
public let myPublicConstant = 0
```

To expose a public variable to allow other code to read and modify its value:

```
public var myPublicVariable = 0
```

To expose an internal variable to allow other code in the same module to read and modify its value:

```
internal var myInternalVariable = 0
```

Expose a fileprivate variable to allow other code in the same file to read and modify its value:

```
fileprivate var myFileprivateVariable = 0
```

To then keep a variable private to prevent other code from accessing it explicitly:

```
private var myPrivateVariable = 0
```

Refactoring code to make it less cyclomatically complex, secure, modular, safe for memory use, and everything else this chapter has covered might be considered technical debt. Technical debt, often likened to financial debt, is a metaphor for the implied cost of future reworking required when choosing an easy but limited solution instead of a better approach that could take more time. Essentially, it's the consequence of prioritizing speed or delivery over optimal, well-designed code. Or more commonly, considering new features over improving on existing code and features. This is fine early in a project,

but can stifle the ability to build new features as the code grows. Determining how much time is spent on technical debt and how much is spent on features, and which features, is one of the responsibilities of a product manager.

Technical quality and product quality are related, but not identical: code can be clean, secure, and well-tested while still failing if it solves the wrong problem for users.

Product Management

Quality isn't just about code, it's also about process. While often shrouded in a veil of mystery, product management is anything but. It's the orchestration of a product's journey, from conception to market domination (or at least, that's the dream!). It's the intersection of business, technology, and user needs, where strategy meets execution, and vision becomes reality.

The product manager is the champion of the product, the voice of the user in the company's ear, and the captain navigating the choppy waters of development. Their core responsibility is understanding the market and user needs. This might mean through research and analysis, or in a team of one, just having some empathy for the people who interact with the projects being built, no matter how large or small. Developers, like product managers become deeply familiar with the landscape, identifying problems and opportunities to solve them.

As product management expands from a team of one and into an organization, some control of what is built is handed off to someone with a focus on the following:

- *Defining the product vision and roadmap*: They craft a clear vision of what the product should be, outlining its features, functionalities, and goals, and then chart the course for getting there.
- *Prioritization and decision-making*: With limited resources, product managers prioritize features and make tough calls, ensuring the right things get built at the right time.
- *Collaboration and communication*: They act as the bridge between various teams, from engineers and designers to marketing and sales, fostering a collaborative environment and ensuring everyone is aligned.

- *Data analysis and iteration*: Product managers are constantly measuring and learning, analyzing data to understand how the product is performing and using it to inform future iterations and improvements.

But product management is more than just a set of tasks; it's a mindset. It requires a blend of strategic thinking, analytical prowess, and creative problem-solving. It demands strong communication skills, the ability to build relationships and influence stakeholders, and a relentless focus on the user experience. Products that don't solve real problems or cater to user needs are destined to fail. Product managers are the gatekeepers of good taste, ensuring that what gets built is not just technically sound but also desirable and impactful. Even in a team of one MacAdmin acting as the developer of tools and the product manager (and tech support), the products should still be impactful given the level of effort put in.

Further, many who have built products go on to be product managers, product owners, or run product organizations. For those considering that future, look at some of these sites for furthering what many would consider the soft-skill side of a developer:

- Product School: `https://productschool.com/`
- Mind the Product: `https://www.mindtheproduct.com/`
- ProductPlan: `https://www.productplan.com/`
- The Product Manager HQ: `https://producthq.org/`

These are training sites, and some even offer certification. But none will help to automate and track what's being done, which can be help to show a future product manager what was done, how long it took, and most importantly, why it was all done. That could be as simple as a spreadsheet, or might involve special software – some of which an organization might already have.

Product Management Software

Product management automation software should free up product managers and developers to focus on the strategic nuances while robots handle the repetitive tasks. Automation can handle tasks like user behavior analysis, competitor tracking, and sentiment analysis, delivering clear and actionable reports to inform decisions. Some of

this is perhaps less impactful for MacAdmins than those selling third-party products – although no one can ever be certain their work won't become a product sold on the open market. Like Zach Halmstad, a founder of Jamf who wrote a tool to automate imaging at the University of Wisconsin-Eau Claire. His Casper Suite evolved into what is now called Jamf Pro.

Most developers just want to write code, not manually send meeting invites to stakeholders, update roadmaps, and chase approvals down all day. Automation tools can streamline these processes, freeing admins to focus on writing great code to automate tasks, keep computers secure, or brainstorm the next killer feature that turns a little software tool to automate admin tasks into a public company. Further, software helps automate customer feedback management, so no need to sift through mountains of user feedback. The software will often categorize and prioritize feedback, making it easier to identify recurring themes and address pain points quickly.

While software can analyze data, it can't truly empathize with users. When a developer is wearing a product manager hat, it's about intuition and human perspective, ensuring products resonate on an emotional level while keeping devices in a state they need to be in and secure. Automation can provide data-driven insights, but ultimately, strategic choices require human judgment and the ability to consider the bigger picture. It might also involve a boss who just needs to check a box on an information security form.

Choosing the right product management software can make a world of difference when it comes to how much time is spent on what an admin acting as a developer in a small team spends on something they may not even want to do in the first place. So here are some of the best product management software tools available, categorized by their specific strengths:

- ClickUp: Best for growing teams, with a comprehensive suite of features for roadmapping, task management, collaboration, and reporting.
- monday.com: Best for ease of use and scalability, offering a visual and intuitive interface that's perfect for teams of all sizes.
- Asana: Best for goal tracking and team alignment, with clear goal breakdowns, progress tracking, and project dashboards.
- Jira: Best for software development teams, with flexible workflows, issue tracking, and agile development support.

- Trello: Best for collaboration and Kanban-style project management, with simple boards and cards for organizing tasks and workflows.
- ProductPlan: Best for beautiful and user-friendly roadmaps, with drag-and-drop functionality and powerful prioritization tools.
- Aha!: Best for enterprise-level roadmapping and strategy, with features like portfolio management and resource allocation.
- Productboard: Best for gathering and prioritizing customer feedback, with tools for collecting user insights and building customer-centric roadmaps.
- UserTesting: Best for conducting user research and testing prototypes, with tools for remote usability testing and video feedback.
- Linear.app: Heavily integrated with AI and systems using AI.

Budgets are tight at most companies and a net-new recurring outlay might not be possible just to track the status of some development project. So if the company already uses a tool like Monday for the marketing organization or Jira for a service desk, then the best tool might just be the one already in use at the organization.

The most important thing is transparency. Let as many people who care know what's being done and why. Every user in an org is a stakeholder, so the more that can join in on the journey, the better.

CHAPTER 8

Using Artificial Intelligence

So far we have focused on learning Swift through doing all the hard work yourself. However, the future of software development is inextricably linked to the use of artificial intelligence. If your journey to learn Swift was predicated on the joy of learning and the desire to build software by hand, you can skip this chapter and get on with using Swift as you see fit. If you're curious about what the future of software development may look like, keep reading.

AI for Coding

Artificial intelligence has had a very profound impact on coding in a very short period of time. While AI has been around in a variety of forms for a number of years, it's only been in the last few that it's become useful enough to speed up the development process.

Early AI for coding was primarily focused on reviewing code that human developers had written looking for bugs and security issues. From there it progressed to being integrated into the development process as an autocomplete tool to fill in verbose chunks of code that the developer would otherwise have to write by hand. Now AI has evolved to where you can "vibe code" just by chatting with an AI agent asking it to do things for you.

The use of AI can be quite controversial. There are very real environmental impacts from the amount of power and water used by the growing number of data centers that are required by the growth of AI services. There are copyright questions based on what materials the AI models were trained on, let alone questions around who owns the output of a model. And then there are the existential questions around what it means for

C. Edge et al., *Beyond Scripting*, https://doi.org/10.1007/979-8-8688-3035-8_8

humanity as AI takes over day-to-day operations. This book does not attempt to answer any of those questions, but it does acknowledge that they exist and are cogent questions to be asking. What this chapter endeavors to do is to prepare you for using AI, if you so desire, with Swift.

Furthermore, we won't attempt to suggest that one AI model is better or worse than another. The AI companies and their models have been leapfrogging each other almost monthly on which model does the best on benchmarks and for specific tasks. In general, most of the current models will do well at building simple to even medium complexity apps for iOS or macOS.

While it's easy to get a sugar high off of coding with AI, keep in mind that you may also get a crash. Developing well-built, secure, performant apps that users want to use still takes a lot of skill. Success with AI coding is often directly linked to knowledge of the coding process, especially as your app becomes more complex. The biggest impact of AI is more the speed at which you can work and how much of the tedium of development you can take out of the process.

Some AI Basics

AI can be quite a challenging topic to understand. Part of this can be because it seems like magic at times, and part of this is because the people building these systems don't always fully understand why they work the way they do. That said, a little background goes a long way toward making you a more effective user of these tools. The better you understand the how and why behind the decisions AI is making for you, the better your code will be.

Large Language Models

The AI systems you'll be using for coding are called Large Language Models, or LLMs. Despite the name, they work with code just as well as they work with human language – to the model, both are just sequences of text.

An LLM is trained by processing an enormous amount of text: books, websites, source code repositories, documentation, and much more. During training, the model learns statistical relationships between words and phrases – essentially learning which things tend to follow other things. The result is a model that can predict what should

come next in a given piece of text. When you ask an LLM a question, what it's really doing is predicting the most useful continuation of the conversation you've started.

This is worth sitting with for a moment. LLMs are not looking things up in a database, and they aren't reasoning through problems the way humans do, at least not in a way we fully understand yet. They are extraordinarily sophisticated pattern-matching systems that can produce output that looks very much like reasoning. This explains both why they can seem impressively capable and why they can sometimes be confidently, completely wrong.

LLMs are primarily very, very good autocomplete systems.

Tokens

LLMs don't process text one character at a time. Instead they work with chunks called *tokens*. A token is roughly equivalent to four characters of English text, or about three-quarters of a word, so a short word like "the" is usually a single token while a longer word might be two or three. Code tokens often break at punctuation boundaries.

Tokens matter practically for two reasons. First, the services you use probably charge based on how many tokens flow in and out of the model. Second, every model has a *context window* – a maximum number of tokens it can hold in its working memory at once. If your conversation or your codebase grows beyond that limit, the model can no longer see everything at once, and its output tends to degrade. Current models have context windows ranging from tens of thousands to millions of tokens, and the number keeps growing, but it's still something to be aware of on larger projects.

Models and Parameters

When you hear that a model has billions of parameters, those parameters are numbers – specifically, weights that were adjusted during training to encode all of those statistical relationships. A model with more parameters has more capacity to capture nuance, but also requires more compute to run. This is why the large cloud-based models are generally more capable than models you can run locally on your Mac, though the gap has been narrowing quickly.

The companies that build these models release different *sizes*, often named something like small, medium, or large, and different *versions* as they improve the underlying training. When you see a version number or a name like "Sonnet" or "o4-mini," you're seeing their way of indicating where a particular model sits in their lineup.

Temperature and Randomness

One characteristic of LLMs that surprises a lot of people is that they aren't deterministic by default. If you ask the same question twice, you may get a slightly different answer each time. This is controlled by a setting called *temperature*. At a low temperature, the model sticks close to the most probable next token at each step, producing consistent and conservative output. At a higher temperature, it introduces more randomness, which can make the output more creative but also less predictable. Most coding tools keep temperature low by default, which is generally what you want when you need the model to produce correct, repeatable code.

What Models Know (and Don't Know)

LLMs have a *training cutoff* – a date beyond which they haven't seen any data. This means a model trained in mid-2024 won't know about APIs introduced in Swift 6.1, or about a security vulnerability discovered last month. When you're working with rapidly evolving frameworks or relying on very recent documentation, be aware that the model may be working from out-of-date knowledge. Many of the AI tools address this by giving the model access to web search or by letting you paste in documentation directly, which is worth doing when currency matters.

Models also only know what's in their training data. If your project uses a private library or an internal API that was never public, the model has no idea it exists unless you describe it or show it the code. The more context you give the model, the better it performs. This is the single most practical insight about working with AI for coding: the quality of your output is largely determined by the quality of your input.

Inference vs. Fine-Tuning

When you use an AI coding tool, you're almost certainly doing *inference* – sending a prompt to a pre-trained model and getting a response back. The model itself isn't learning from your conversation in any durable way; each session generally starts fresh.

Fine-tuning is the process of taking a base model and continuing to train it on a specialized dataset to make it better at a specific domain. Some companies fine-tune models specifically on high-quality code. You may encounter tools that claim

to let you fine-tune a model on your own codebase, which can improve performance on proprietary patterns and conventions. This is a more advanced topic than most developers need to think about when starting out, but it's good to know the term when you encounter it.

Xcode and AI

Many developers building in Python, Go, Rust, or other languages use a standalone IDE (Integrated Development Environment) like Microsoft's Visual Studio Code https://code.visualstudio.com. VS Code is free, has a wealth of third-party plugins, and can work with a lot of different languages. It also has an AI component provided by GitHub Copilot, another Microsoft product. Cursor https://cursor.com is a fork of VS Code with a proprietary AI engine added into it. Claude Code https://claude.ai/code is a terminal-based agentic coding assistant from Anthropic that works alongside any editor. Another option is ChatGPT https://chatgpt.com/, which is a model that can be used with an IDE through a variety of plugins, and also has its own desktop and mobile applications.

All of these tools, and others, can write Swift, but they probably aren't the best option as they don't support previewing SwiftUI interfaces, for example. However, you may be able to use a subscription you already have for these tools with Xcode.

Apple began adding AI features to Xcode during the summer of 2024 with Xcode 16. Apple's initial foray was to allow expanded code autocomplete using local models on Apple Silicon systems. Apple was somewhat unique in both by shipping a local model so you could access AI features even when offline, and by not charging a subscription for this.

However, autocomplete was just a fraction of what developers had been using with some of the commercial AI providers like Cursor and Claude. While there were some options to use external AI as a plugin, it was clear that Apple needed to add the commercial AI models in as first-class citizens.

Starting with Xcode 26, which was released during Apple's WWDC (Worldwide Developers Conference) in the summer of 2025, you can now add your accounts for Claude or ChatGPT directly into Xcode in the Settings as in Figure 8-1.

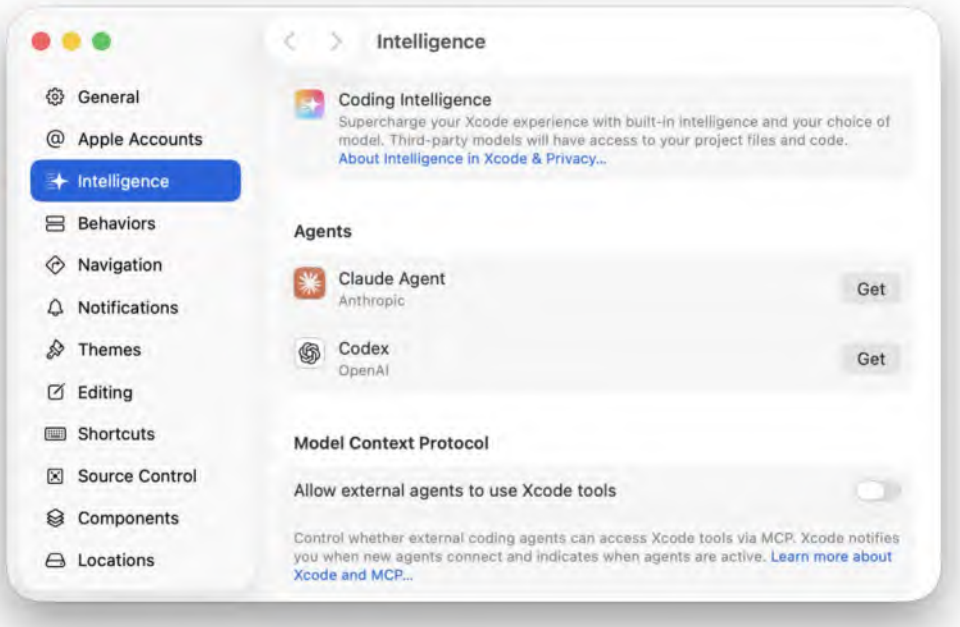

Figure 8-1. *Xcode Intelligence settings*

You can add additional models, either local or remote, by using the "Add a Chat Provider..." button and configuring the remote or local URI and any authentication (Figure 8-2).

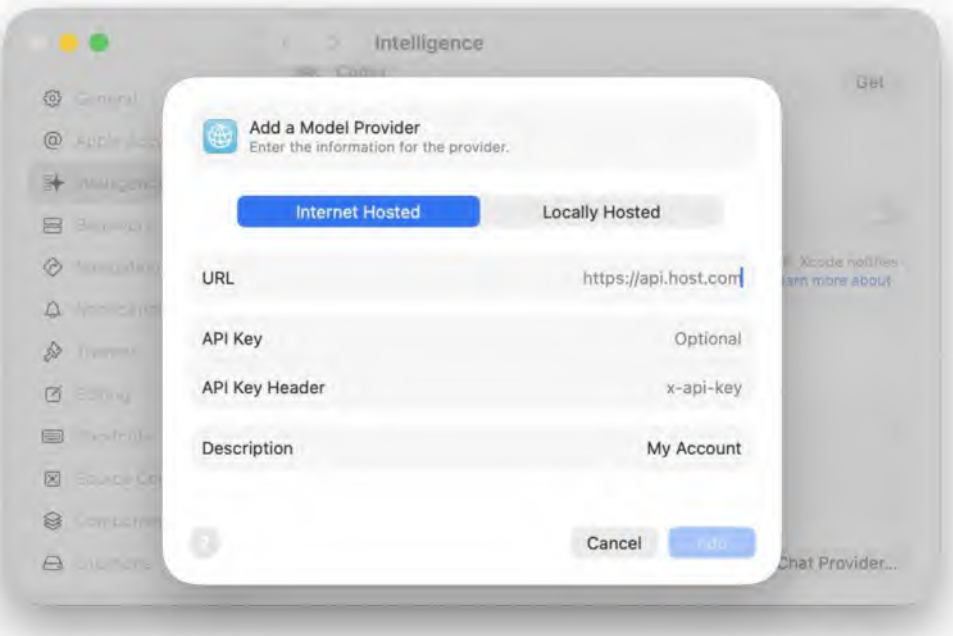

Figure 8-2. *Setting up a model for a chat provider*

Once configured, you can use the "Coding Assistant" star icon in the upper left-hand corner of the Xcode window, as seen in Figure 8-3 to open up a new chat.

Figure 8-3. *Xcode will show a Coding Assistant icon in the window's menu bar once configured*

If you are using a model with an agent, you may need to download that, but Xcode will help you through the process.

You will most likely need to have a subscription to use a third-party model. Some of the models may offer limited free options, so look at what's available. The pricing and usage limits of the services have been evolving rapidly, so check online at what the current situation is.

Agents vs. Chat

In Xcode, you have the option of connecting to AI via an Agent or chat. Using an agent means Xcode is connecting to local tooling from the AI vendor, while connecting via Chat means Xcode is making API connections to the AI service's cloud provider. The difference is that Agents require a download in most cases, although Xcode will facilitate this, and Agents have access to local configurations and other tools. In general, you probably want to use the Agent method if you have a subscription with those providers as it will be more powerful.

Vibe Coding

If you are learning Swift because you have an idea you want to get out of your head, but never had enough coding knowledge to feel like you could make it happen, this is going to be an amazing journey for you.

While the term "vibe coding" can have some negative connotations, it's actually a great way to get your feet wet with what AI can do for you. While this isn't advised for any production code, it's becoming more common for proofs of concept, mockups, and other projects that need to be done quickly.

The general gist of vibe coding is to just chat with the agent, give it broad strokes about what you want to do, and then let it go. As models have gotten better, vibe coding a mostly working app can be a real rush, especially to a non-traditional developer. In a matter of minutes, you can have a generally decent application that you can then further refine your thinking around what you are looking to build.

To start with, create a new project in Xcode, then go to the Coding Assistant icon to open up the AI chat window. Then just ask the AI to make an app for you.

```
Build an iOS app to play Sudoku. In particular focus on being able to add
pencil marks to cells and easily see patterns from the pencil marks.
```

Is a prompt you can try if you don't have any ideas.

Prompts are written in plain language, and to start at least, don't have to follow any particular syntax. As you get more advanced in the use of AI within an IDE, there are a number of strategies you can use to make the AI more efficient. We'll cover some of these later in this chapter.

As the application builds, you may be asked to approve various tool calls. This is a protection mechanism to ensure that you don't get a runaway AI trying to do too much. While you can create an allowlist to not be asked to approve as much, you may want to approve each call just to get an idea of what the AI may be doing under the hood.

Depending on your model subscription, you may not have enough tokens to finish the application in one sitting. However, you should quickly see the power of what the AI is doing with your code. Once the AI has finished, feel free to edit the code if you want or go back into the chat and ask the AI to change any aspect of what it's already built.

It's probably a good idea to use a version control system like `git` as you use AI, as you may want to frequently undo what the AI has done if it gets ahead of itself or goes down a path you weren't expecting.

Writing Tests

A fantastic use for AI is to help you write tests for your code.

Many developers loathe writing test cases for their code. Testing strategies can be complicated to do well, and they aren't nearly as fun as writing new code.

Using AI to write tests for you, especially as a beginning developer, can take a lot of the grunt work out of test writing and immediately make for better code. Even if you want to write all of the code yourself as you learn, having the AI write your tests is like having a colleague check your code. Just remember to review the tests created by AI; a bad test can create havoc down the line in subtle ways that may take hours to find.

Advanced Strategies for AI

Once you move beyond the vibe coding aspect of chatting with AI, you will probably want to start seeing if you can do real work with it. To that end, here are a few strategies that will help you wrangle the AI to do more of what you need.

Plan First

Some of the models have an official planning mode now, but all of them should understand what to do when you explicitly ask the model to plan out the next steps and write them down.

Models do best when the amount of context the model has to understand is well-managed. In this case, by creating a plan, the AI can then work through the individual pieces of the plan without having to keep all of the application in context at the same time. As models, and the tooling that works with them, have progressed, more and more planning is being done; however, you can ensure this is the case by asking for a plan first.

The plan will be written out into the project as a Markdown file. This gives you an opportunity to read through the plan and correct anything or provide more context. Feel free to add links to external web pages or other sources that better clarify what you want to do. When the AI begins working the plan, all of that can come in handy to make sure it knows what you want.

It's also common for the models to ask questions during this planning phase. The model may want to get clarification on a specific path to take, or ask what color palette you'd like to use. Don't be shy answering the questions. The more information you give, the better the application you'll typically get.

Product Requirements Document

In professional development environments, it's very typical for a Product Manager to draft up a list of product requirements that is then used in discussion with the development team to plan out how an application should be built and what it should do. You can use a similar flow with the AI even if you're a team of one.

Building out a Product Requirements Document (PRD), which is essentially just a much more detailed plan than what the AI would build on its own, can help you better understand what you want the application to do.

PRDs don't have a specific format and often include different sections depending on what type of application is being built, so you can easily make this very unique to your own specific circumstances.

Also, feel free to have the AI interview you! The model should be able to help build out the structure of the PRD simply by asking it to help you.

Spec-Driven Development

An idea that has been around since the 1960s, Spec-Driven Development (SDD) is a method for using a machine-readable specification that is then used to ensure that the final application does what is expected of it.

SDD is very well suited to AI development because the AI can use the spec both to understand how to build the application and to test it when it's done.

For the most part, SDD mainly means getting even more in depth with the PRD and adding sections such as user feedback, testing requirements, cost estimations for any cloud services that are in use, and a variety of other information. Again, a lot of the work for ensuring the AI does the right thing is best done up front, and SDD fits into that flow very well.

There are a wide variety of SDD methodologies out there. Many of them have cute names. Here are a few that you can start with:

- *BMAD*: Breakthrough Method for Agile AI-Driven Development – https://github.com/bmad-code-org/BMAD-METHOD
- *Ralph*: an autonomous AI agent loop that runs repeatedly until all PRD items are complete – https://github.com/snarktank/ralph

Agents and Skills

Many of the AI tools have support for both agents, where the AI can spin off subprocesses to work on just part of the project, or skills, which are Markdown files that tell the AI how to do something. Unfortunately, Xcode doesn't currently have much support to use these, with the exception of Claude and OpenAI. However, you are more than welcome to start a project in VS Code, Cursor, Claude, or any other IDE that does support agents and skills and then move over to Xcode to do the build.

Agents can help with building PRDs. For example, the BMAD SDD method has a collection of agents that can act as end users for your application. BMAD will have a virtual roundtable where the individual agents take on the persona of various users and provide feedback on what you are building. It will even create full user stories for you, if you would like, that plot out what a user is trying to do in your application.

Skills can be written to explicitly inform the AI what to do for various tasks. It's not uncommon to have tasks such as working with a specific API or deploying the code into a cloud service captured in a skill. The main point of the skill is to keep the AI from having to guess or come up with its own idea of what to do. Using a shared set of skills across a dev team will greatly help with consistency.

MCPs

Model Context Protocol (MCP) is a method of accessing a service via AI. Currently, this is another area where Xcode has limited functionality, but other IDEs are becoming quite advanced.

For example, Jira has an MCP that allows IDEs to access a ticketing system you may be using to keep track of the development process. As the AI works on various parts of the project, it can use the Jira MCP to read the ticket it should be working on and then update the tickets as the AI makes progress.

A growing number of SaaS applications have MCPs that can easily be plugged into IDEs. Apps ranging from Figma, where you may be doing your UX designs, to HubSpot, to check for any customer support tickets, to Slack for team communication, to GitHub for pulling in issues and pull requests – all of these can be wired directly into your AI workflow. Rather than copying and pasting information between tools, the AI can reach out and read what it needs on its own.

The MCP ecosystem is still young and the available integrations change frequently. A good place to start is by looking at the MCP server directory for whatever AI tool you are using, as most of the major ones now maintain a curated list of verified integrations.

Xcode doesn't have much support for connecting to MCPs, although if you've set up an Agent in Xcode, that Agent may have access to MCPs configured outside of Xcode.

Alternatively, you can set up Xcode to be an MCP for other AI tools (Figure 8-4). You can enable this via the "Allow external agents to use Xcode tools" in the Intelligence section of Xcode's Settings.

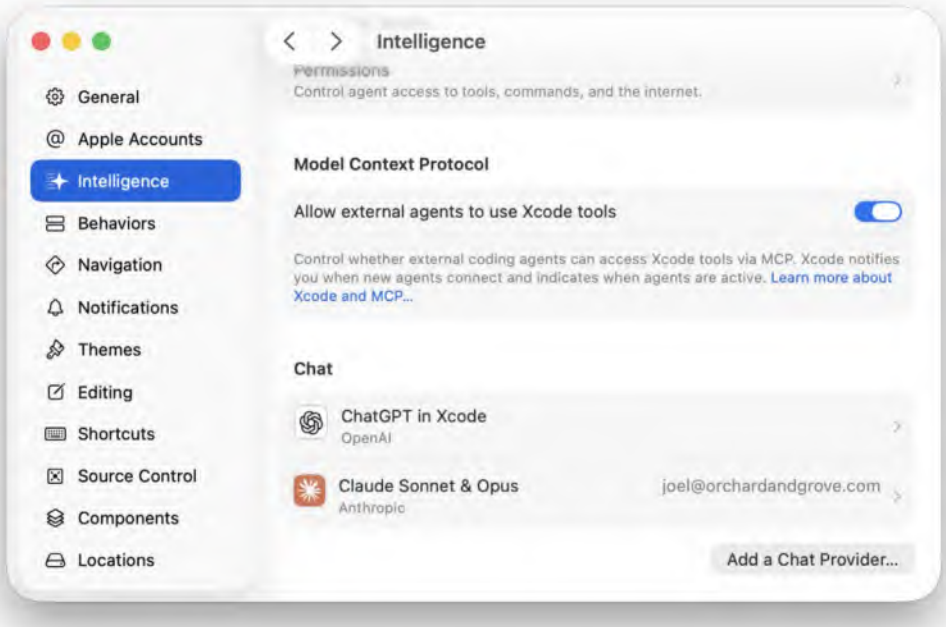

Figure 8-4. *Xcode Intelligence settings for enabling MCP connections*

Once set up, other tools can use Xcode as a local MCP for the purposes of building the applications and working with the Xcode project file. See the Apple documentation for how to connect your tools to it: `https://developer.apple.com/documentation/xcode/giving-external-agents-access-to-xcode`.

Don't Be Stupid

It can be all too easy to make bad decisions when working with AI. Even if it does make the AI run faster, it's a bad idea to hand over API keys, secrets, and other privileged information like customer data that you might have to the AI. Social media is full of stories about AI deleting every database in a company or "helpfully" exposing company secrets to the Internet because it made processing that information that much faster.

Unless you've really vetted the model providers that you're using and have signed agreements with them, think twice before exposing proprietary code that might be used by the model to learn off of. Most larger organizations go to great lengths to only allow approved AI for reasons like this.

When AI Goes Wrong

AI coding tools are impressive, but they fail in predictable ways. Knowing what to watch for – and how to recover – will save you a lot of frustration.

The Doom Loop

The most common failure mode is what developers call a doom loop: the AI tries to fix a bug, introduces a new one, then tries to fix that, and gradually makes the code worse with each pass. You'll recognize it when the AI starts apologizing in its responses, when the same error keeps appearing under slightly different forms, or when the code is growing in size but the app still doesn't work.

The right move when you spot this early is to stop the AI and start a fresh conversation. Don't ask it to "try again" or "go back to what you had before" in the same context window – the model has already seen too many failed attempts and tends to keep pulling from that polluted history. A clean context with a clear, focused prompt will almost always outperform continuing a broken conversation.

Commit Early and Often

The single most practical habit you can develop when working with AI is to commit your code to `git` frequently. Before you ask the AI to make any significant change, make a commit. That way, if the AI goes off the rails, you can simply reset to your last known good state rather than trying to manually untangle what it did.

```
git add -A && git commit -m "working state before AI refactor"
```

This is even more important during vibe coding sessions, where the AI may touch a large number of files in a single pass. A commit takes seconds and can save you an hour of recovery work.

Hallucinated APIs

LLMs will sometimes confidently write code that calls an API that doesn't exist, uses a method signature that was deprecated two versions ago, or invents a parameter that was never part of the framework. This happens most often with less common frameworks, very recent APIs that postdate the model's training data, or private internal APIs the model has never seen.

The tell is usually a compiler error about a missing method or a type that can't be found. When you see this, don't ask the AI to fix the compile error – it will often dig deeper into the hallucination. Instead, look up the correct API in Apple's documentation directly, then tell the AI specifically what the correct method or type is and ask it to revise.

Context Overload

On larger projects, conversations grow long and the AI starts losing track of decisions made earlier in the session. You may notice it reverting changes it already made, contradicting an approach it agreed to earlier, or producing output that seems to ignore half the requirements.

This is the context window filling up. The fix is to start a new conversation and open with a concise summary of where the project stands and what you need next, rather than continuing to append to a conversation that has grown unwieldy. If you have a plan or PRD document in your project, pointing the AI at that file at the start of each new session is a good way to restore context efficiently.

When to Take Back Control

There will be times when the right answer is to stop using the AI for a particular problem and write the code yourself. Complex concurrency bugs, subtle memory management issues, and intricate state machine logic are all areas where an AI can lead you further astray than if you had just worked through it manually. The AI is a tool, not a replacement for understanding your code. The more of this book you have worked through, the better equipped you are to recognize when that line has been crossed.

Your Development Journey

As Uncle Ben said, "With great power comes great responsibility." AI is no exception to this rule. If you decide to use AI, it can be an amazingly powerful tool to help you write applications. In fact, you can even use AI to write applications in languages you've never learned!

On the other hand, Xcode isn't forcing you to use AI, and writing Swift applications by no means requires the use of AI.

Hopefully, this chapter has helped you become more aware of how AI can be used in the development process and to better decide how you want to use it in your development journey.

Index

A

C. Edge et al., *Beyond Scripting*, https://doi.org/10.1007/979-8-8688-3035-8

B

C

D

E

F

G, H

I

J

K

L

M

N

O

P

Q

R

S

T

U, V

W

X, Y, Z