PREVIEW

The Rails and Hotwire Codex

Build an app for web, iOS, and Android

Ayush Newatia

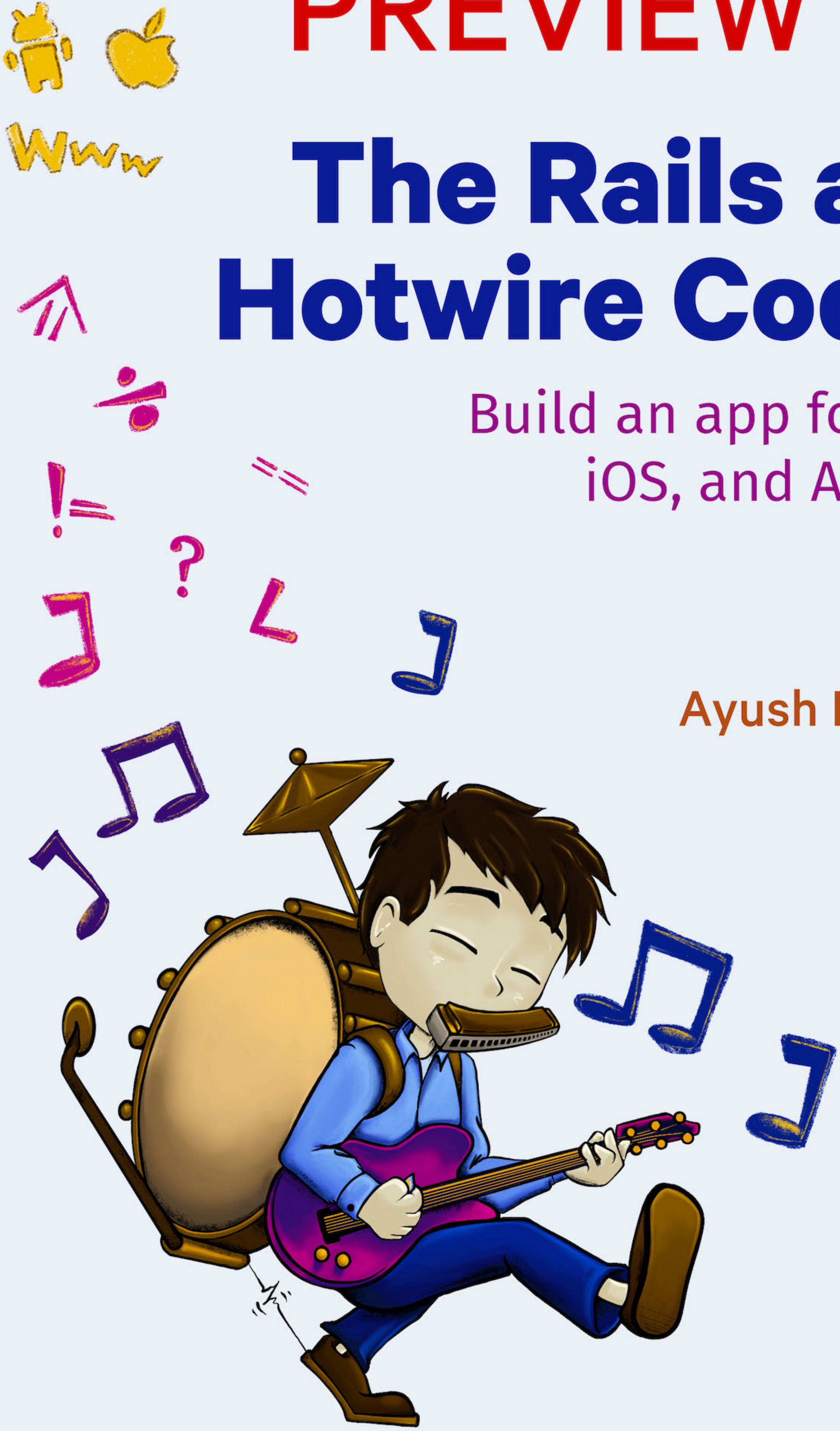


Table of Contents

Introduction

It’s 2024. Building an app solely for The Web doesn’t quite cut it any more. iOS and Android apps are essential for a successful SaaS product.

Hybrid apps are unpopular because they are believed to degrade the user experience. While that may have been accurate in the past, mobile performance has now reached a point where hybrid apps, when done right, can be truly delightful.

There’s also the massive advantage of writing the views once, for web, and then reusing them in the native apps. This is a huge competitive edge.

Hotwire provides the tools to build high-fidelity web apps without a monolithic front-end framework like React. It also gives us native extensions using which we can build iOS and Android *wrappers* for web apps, with the ability to go fully native when required. In this book, we’ll build an application for Web, iOS, and Android without losing our sanity!

How to read this book

I recommend *working through* this book rather than reading it to get the most out of it. Keep this book open alongside your development environment and follow along by typing out the code instead of copying it across. Your mileage may vary though and it’s best to follow whatever method you’re comfortable with.

If you’re not interested in the native side of things, you can skip the chapters and sections related to iOS and Android. They build on top of the web app so you won’t miss anything. The iOS and Android apps are totally independent of each other as well, so you can choose only one of those platforms if you wish. For example, you might want to skip iOS if you haven’t got a Mac.

Prerequisites

This book assumes a basic understanding of programming for the web, object-oriented programming, Ruby, Rails, modern JavaScript, HTML, CSS, and version control using Git. If you're familiar with basic OOP principles like classes and inheritance, setting up Active Record models and associations, defining Rails routes to connect to a controller action, and rendering a view from a controller action; you'll be able to follow along just fine.

If any of the above terms didn't make sense and you need to get up to speed, I recommend completing The Ruby on Rails Tutorial by Michael Hartl before starting this book.

The iOS app will be written in Swift and the Android app in Kotlin. You don't need any knowledge of these languages or the native APIs. We'll cover all that in this book. Although any familiarity with these languages will be beneficial!

Hardware and Software

iOS apps can only be developed on a Mac. If you want to work through the native iOS sections, you'll need a Mac with Xcode installed. For the Android app, you'll need Android Studio which works on Mac, Windows, and Linux.

You'll also need Ruby 3, Rails 7, Node.js 16 and PostgreSQL 14 installed on your local machine. We won't be covering how to set up your local development environment.

I recommend using a Mac or Linux machine to work through this book. If you're using a Windows machine, installing an Ubuntu VM might be worth doing.

What we’re going to build

We’ll build a neighbourhood marketplace app à la Gumtree or Craigslist. We’ll call it **Piazza**. The bulk of the work will be on web, and we’ll use Hotwire’s native extensions for the iOS and Android apps.

The web app will need to be fully responsive all the screens will be reused in the mobile apps. Since responsive design could be a book in its own right, we’re going to use a highly technical concept known as cheating. Bulma is a CSS framework and UI Kit that gives us all we need so we’ll use that to ensure our app looks great and is responsive!

Let’s dive in!

Chapter 1. Getting started

First things first, we need to create repositories for the Rails app and native apps. We'll create individual repositories for all 3 to keep their sizes in check.

1.1. The prerequisites

Verify that the right versions of Ruby and Rails are installed. You'll need at least Rails 7.1 and Ruby 3.2 for this book.

```
$ rails -v
Rails 7.1.3.2
```

```
$ ruby -v
ruby 3.2.2 (2023-03-30 revision e51014f9c0) [arm64-darwin22]
```

If you've got older versions installed, you can install Rails 7 using:

```
$ gem install rails -v 7.1.3.2
```

To manage your Ruby version, I recommend setting up rbenv.

Node.js and Yarn are required to transpile JavaScript and install frontend libraries.

```
$ node -v
v20.1.0
```

```
$ yarn -v
1.22.19
```

You'll need Node.js v16 or newer to avoid issues with this project. Installation instructions can be found at https://nodejs.org/en/download/. If you haven't got Yarn installed, run the following command after installing Node.

```
$ npm install --global yarn
```

Ensure you have PostgreSQL and Foreman (used to orchestrate multiple processes in development) installed.

```
$ foreman -v
0.87.2

$ postgres --version
postgres (PostgreSQL) 14.2
```

Foreman can be installed by running:

```
$ gem install foreman
```

If you don't have PostgreSQL installed, do a search using your favorite search engine for installation instructions for your platform. If you have Homebrew installed, you can run:

```
$ brew install postgresql
```

1.2. Creating the Rails app

In the Rails app, we'll use ESBuild to transpile and bundle JavaScript[1] and Bulma for CSS. Propshaft will be used to deliver assets. The database will be PostgreSQL both locally as well as in production.

Why transpile and what does Propshaft do?

Transpiling JavaScript allows us to use next-generation features that aren't yet available in all browsers. This means we can write code that's more concise and readable with the trade off that we have an additional *build* step. Given the ease of use of ESBuild, and the fact that we mostly won't even notice it's there, I feel this is a worthy tradeoff.

Vanilla CSS can be tedious to write which is why preprocessors like Sass, Less, and PostCSS have cropped up over the years. Using a preprocessor, writing CSS becomes simpler and DRYer before it's transpiled into syntax that browsers can understand. We're using the Bulma library for reasons stated earlier and that's written in Sass, so we'll use the Sass preprocessor in Piazza.

Finally, Propshaft. Before an asset is delivered to the browser, it should be *stamped* with a *digest*. This means it's run through a hash function to produce a unique string (or *digest*) which is then added to the filename. This way, assets can be cached aggressively because the filename changes when the asset changes, meaning the latest asset is always retrieved.

You may be acquainted with the incumbent Sprockets powered asset pipeline in Rails. Propshaft is a simpler alternative designed specifically to *deliver* assets rather than transpile them as well.

Run the command below to create the Rails app:

```
$ rails new piazza -j esbuild --css bulma -a propshaft -d postgresql
```

This will create the app in a folder called `piazza`. We'll also be creating apps called **piazza** on Android and iOS. To allow us to keep all 3 apps in the same folder, rename the folder containing the Rails app.

```
$ mv piazza piazza-web
```

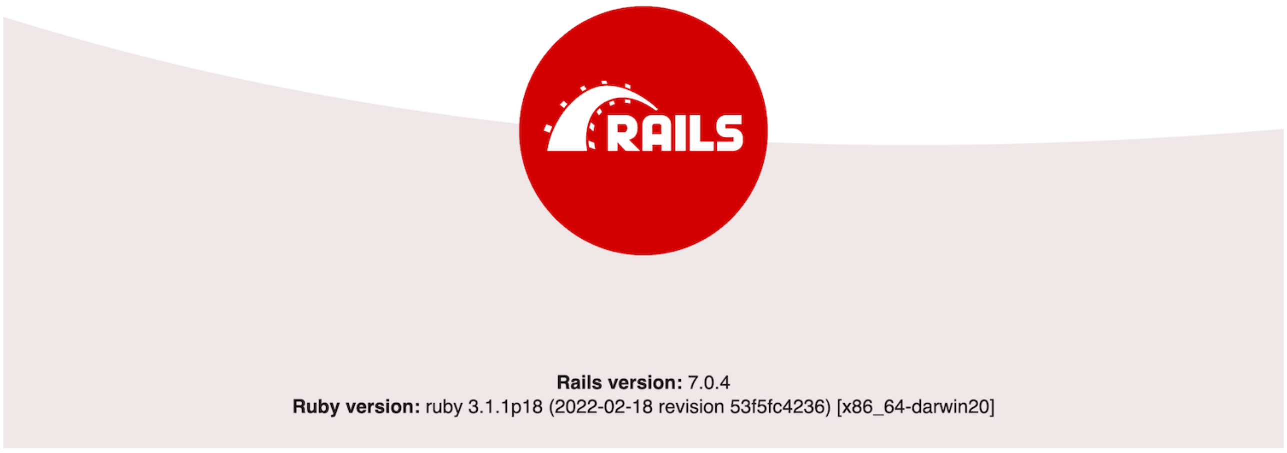
Now we can start the app.

```
$ cd piazza-web
$ bin/rails db:prepare
$ bin/dev
```

If something goes wrong while preparing the database, you may not have PostgreSQL running on your local machine so it's worth checking that.

`bin/dev` uses Foreman to run the Rails server and processes to watch the CSS and JS files so they're recompiled automatically when changed. The commands to start these processes are defined in `Procfile.dev`.

The Rails server should now be running and you'll see the page in Figure 1 at http://localhost:3000.



Rails version: 7.0.4
Ruby version: ruby 3.1.1p18 (2022-02-18 revision 53f5fc4236) [x86_64-darwin20]

Figure 1. The default Rails welcome page

1.3. Pushing to GitHub

It's wise to host our source code on a cloud VCS provider. It acts as a backup in case something goes wrong with our development machine and also makes it easy to trigger deployments to production.

GitHub is the most popular code hosting provider. Create an account on GitHub or login if you've already got one. Create a new repository called `piazza-web`.

Once you've done that, follow the on screen instructions to *push an existing repository from the command line*.

1.4. Deploying to production

Now that we have a functioning Rails app on GitHub, let's deploy it to production. It's good practice to deploy frequently so any problems are caught as soon as they're introduced.

We'll be deploying to Render.com which is a Platform-as-a-service hosting provider similar to Heroku. It provides several benefits over Heroku such as HTTP/2[2] and has a great free tier[3].

Render has the concept of *services*. Each app comprises of one or more types of service. We'll need a web service and a database to deploy our app. Both these service types are available under the free tier.

If you're using a Mac, the app needs to be Linux enabled by running:

```
$ bundle lock --add-platform x86_64-linux
```

1.4.1. A production config for Puma

The default configuration for the Puma web server that Rails gives us out of the box is fine for development. In production, some changes are needed to improve performance.

Make a copy of the default Puma configuration.

```
$ mkdir config/deploy
$ cp config/puma.rb config/deploy/puma.rb
```

Copy the contents of Listing 1 into the new config file. The highlighted lines show the changes the default config.

*Listing 1. The production Puma configuration (**`config/deploy/puma.rb`**)*

```
# Puma can serve each request in a thread from an internal thread
pool.
# The `threads` method setting takes two numbers: a minimum and
maximum.
# Any libraries that use thread pools should be configured to match
# the maximum value specified for Puma. Default is set to 5 threads
for minimum
# and maximum; this matches the default thread size of Active Record.
#
max_threads_count = ENV.fetch("RAILS_MAX_THREADS") { 5 }
min_threads_count = ENV.fetch("RAILS_MIN_THREADS") {
max_threads_count }
threads min_threads_count, max_threads_count

# Specifies the `worker_timeout` threshold that Puma will use to wait
before
# terminating a worker in development environments.
#
worker_timeout 3600 if ENV.fetch("RAILS_ENV", "development") ==
"development"

# Specifies the `port` that Puma will listen on to receive requests;
# default is 3000.
port ENV.fetch("PORT") { 3000 }

# Specifies the `environment` that Puma will run in.
#
```

```
environment ENV.fetch("RAILS_ENV") { "development" }

# Specifies the `pidfile` that Puma will use.
pidfile ENV.fetch("PIDFILE") { "tmp/pids/server.pid" }

# Specifies the number of `workers` to boot in clustered mode.
# Workers are forked web server processes. If using threads and
workers together
# the concurrency of the application would be max `threads` *
`workers`.
# Workers do not work on JRuby or Windows (both of which do not
support
# processes).
#
workers ENV.fetch("WEB_CONCURRENCY") { 4 }

# Use the `preload_app!` method when specifying a `workers` number.
# This directive tells Puma to first boot the application and load
code
# before forking the application. This takes advantage of Copy On
Write
# process behavior so workers use less memory.
#
preload_app!

# Allow puma to be restarted by `bin/rails restart` command.
plugin :tmp_restart
```

We need to tell the app where to find the production database. Open `config/database.yml` and scroll down to the `production` key which should look like Listing 2

Listing 2. The existing production database configuration (**`config/database.yml`**)

```
# ...

production:
  <<: *default
  database: piazza_production
  username: piazza
```

NOTE For simplicity and brevity, code snippets will sometimes have a commented ellipsis as seen in Listing 3 (`# ...`). This signifies omitted code not currently relevant and should not be copied literally.

Change the lines in Listing 2 to Listing 3.

Listing 3. The Render production database configuration (`config/database.yml`)

```
# ...

production:
  <<: *default
  url: <%= ENV['DATABASE_URL'] %>
```

Before the app is deployed, the database migrations need to be run and JavaScript and CSS compiled . Let's write a script to do that.

```
$ touch bin/render-build.sh
```

Fill it in with Listing 4.

Listing 4. The build script (`bin/render-build.sh`)

```
#!/usr/bin/env bash
# exit on error
set -o errexit

bundle install
bundle exec rake assets:precompile
bundle exec rake assets:clean
bundle exec rake db:migrate
```

We also need to make the script executable.

```
$ chmod a+x bin/render-build.sh
```

Lastly, we need to define the Render services needed to run our app. We can do this in code using Render's YAML specification for services[4] known as *Blueprints*.

Create a file in the project root to define the infrastructure.

```
$ touch render.yaml
```

Define a web service and database on the free tier as demonstrated in Listing 5.

Listing 5. Define the services needed for Piazza (`render.yaml`*)*

```
services:
  - type: web
    name: piazza-web
    env: ruby
    plan: free
    numInstances: 1
    buildCommand: ./bin/render-build.sh
    startCommand: bundle exec puma -C config/deploy/puma.rb
    envVars:
      - key: DATABASE_URL
        fromDatabase:
          name: piazza-db
          property: connectionString
      - key: RAILS_MASTER_KEY
        sync: false

databases:
  - name: piazza-db
    plan: free
    postgresMajorVersion: 14
```

The contents of Listing 5 should be self-explanatory. We're defining a web service along with a build and start command, and a database.

Commit and push the code so we can deploy it.

```
$ git add .
$ git commit -m "Set up Render deployment"
$ git push
```

Create an account at Render.com or log in if you already have one. Once logged in, go to your Account Settings as shown in Figure 2.

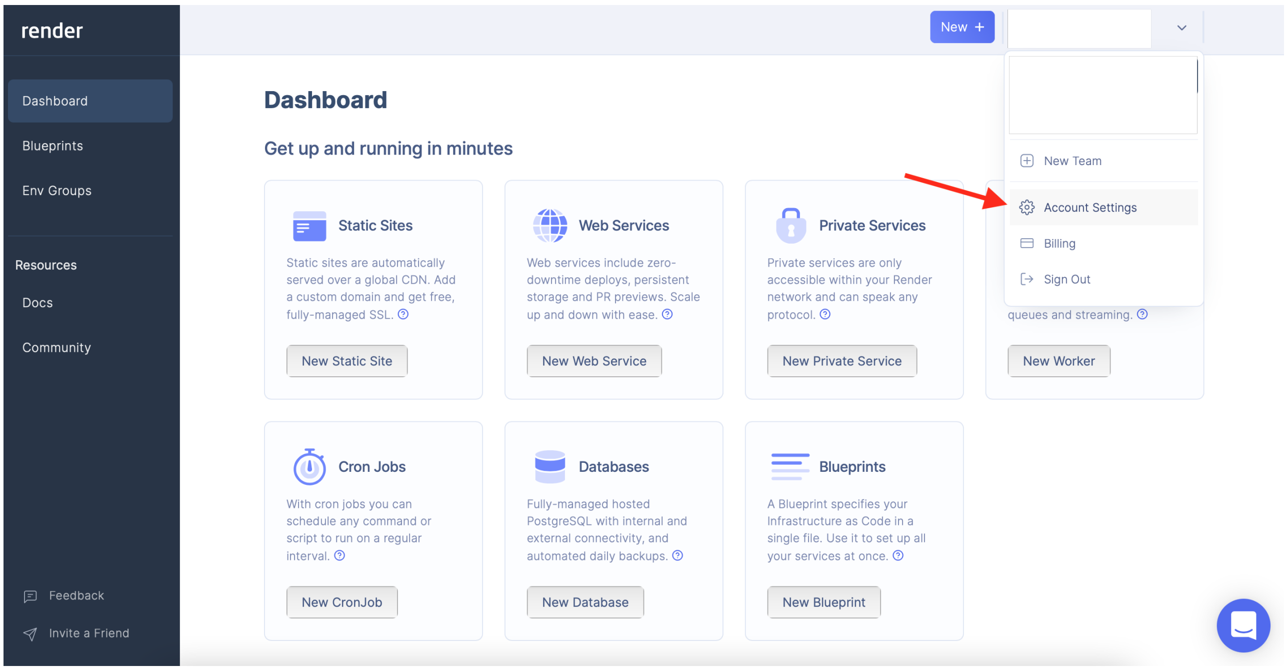

render
New +
Dashboard
Blueprints
Env Groups
Resources
Docs
Community
Feedback
Invite a Friend
Dashboard
Get up and running in minutes
New Team
Account Settings
Billing
Sign Out
Static Sites
Static sites are automatically served over a global CDN. Add a custom domain and get free, fully-managed SSL.
New Static Site
Web Services
Web services include zero-downtime deploys, persistent storage and PR previews. Scale up and down with ease.
New Web Service
Private Services
Private services are only accessible within your Render network and can speak any protocol.
New Private Service
queues and streaming.
New Worker
Cron Jobs
With cron jobs you can schedule any command or script to run on a regular interval.
New CronJob
Databases
Fully-managed hosted PostgreSQL with internal and external connectivity, and automated daily backups.
New Database
Blueprints
A Blueprint specifies your Infrastructure as Code in a single file. Use it to set up all your services at once.
New Blueprint

Figure 2. Go to your Render account settings

On the account settings page, connect your GitHub account.

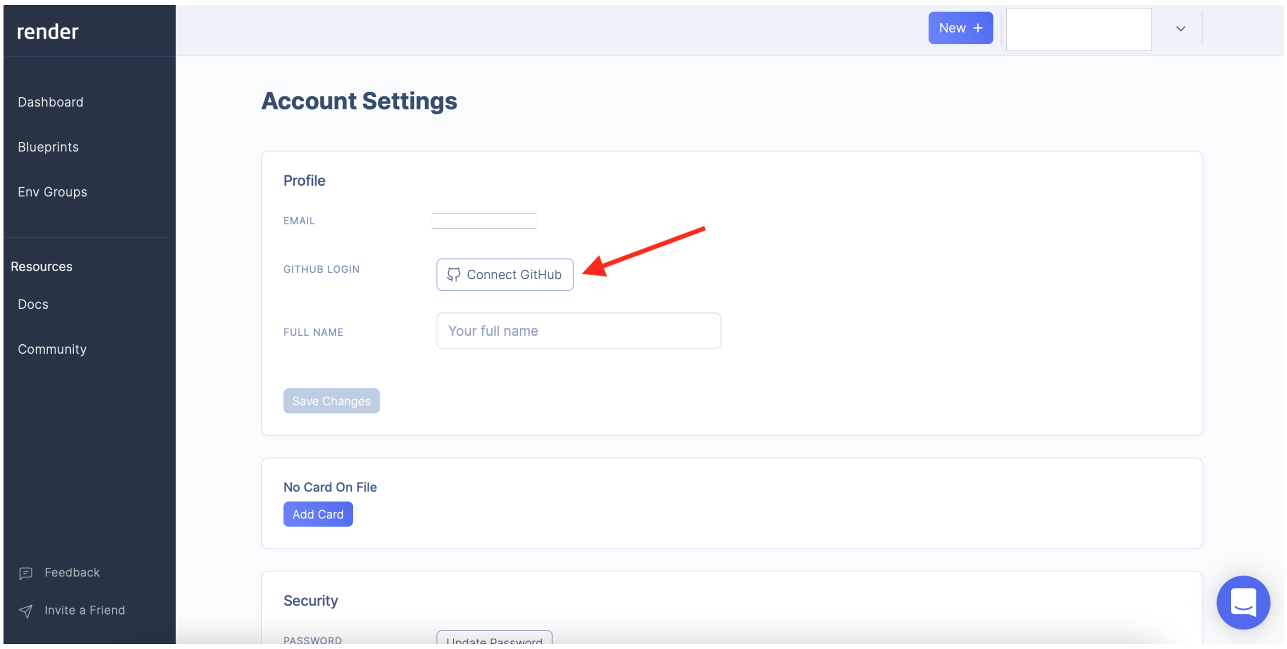
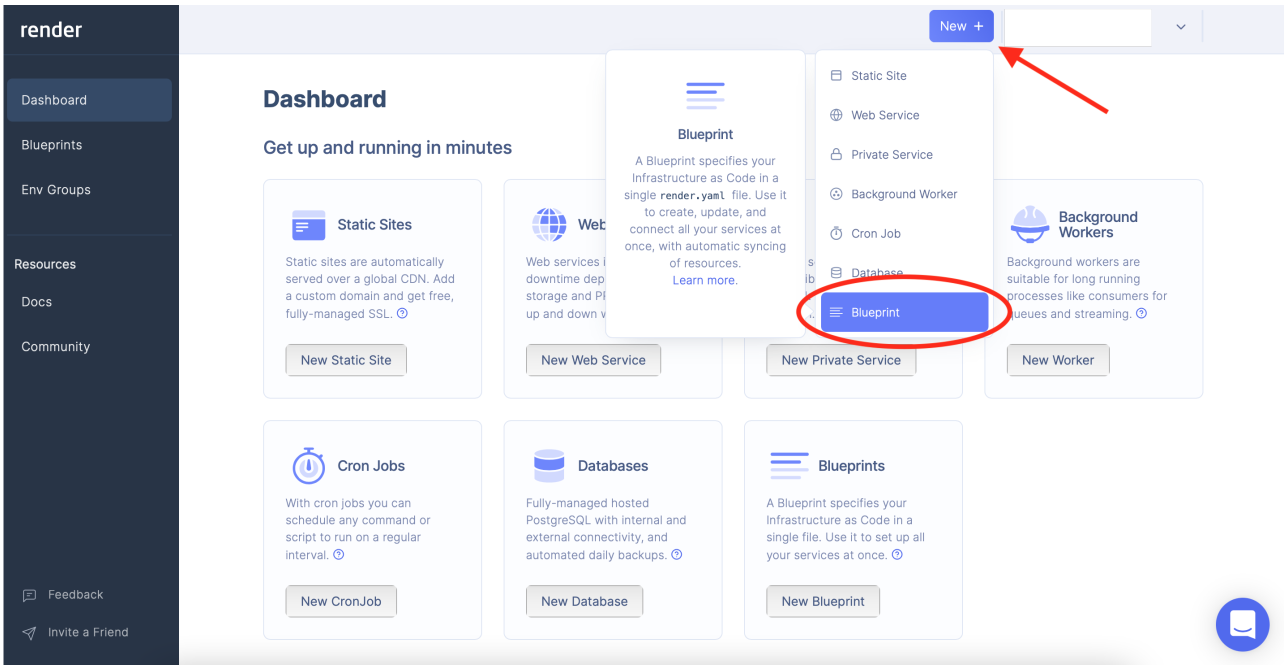


Figure 3. Connect Render to GitHub

After GitHub's been connected, we can point Render to our repository and blueprint. Create a new blueprint as shown in [Figure 4](#).



A Blueprint specifies your Infrastructure as Code in a single `render.yaml` file. Use it to create, update, and connect all your services at once, with automatic syncing of resources.

Static sites are automatically served over a global CDN. Add a custom domain and get free, fully-managed SSL.

Web services i… downtime dep… storage and Pl… up and down v…

Background workers are suitable for long running processes like consumers for queues and streaming.

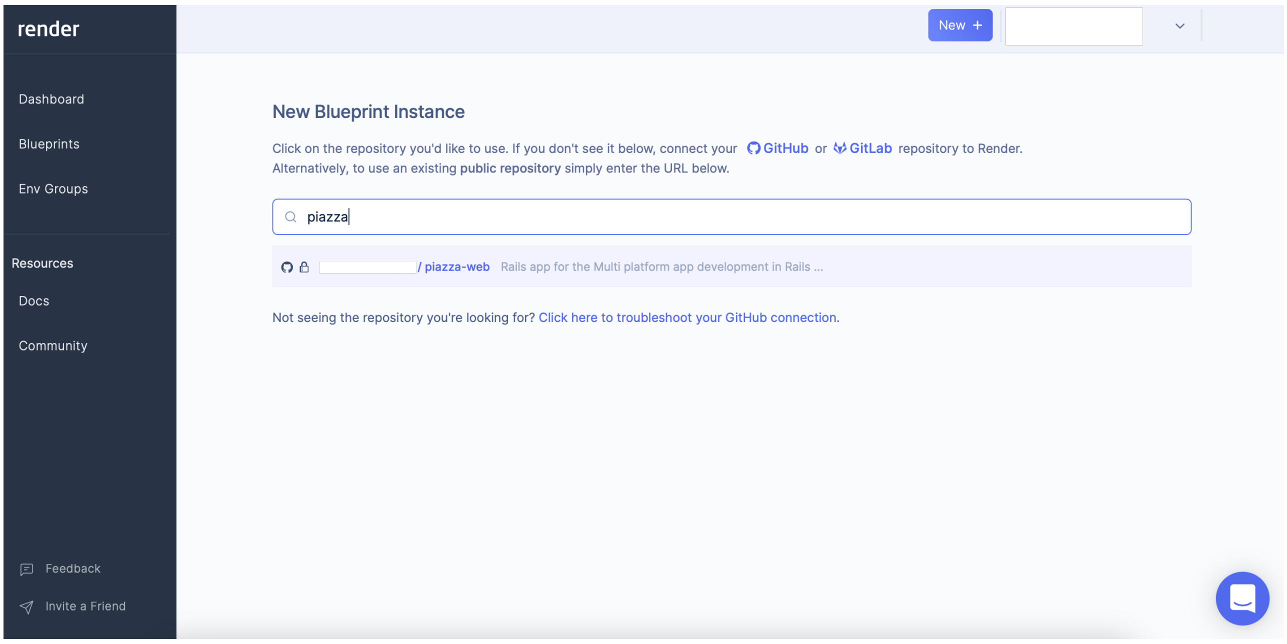
With cron jobs you can schedule any command or script to run on a regular interval.

Fully-managed hosted PostgreSQL with internal and external connectivity, and automated daily backups.

A Blueprint specifies your Infrastructure as Code in a single file. Use it to set up all your services at once.

Figure 4. Create a new Blueprint on Render.com

You'll then see the page in [Figure 5](#). Find your `piazza-web` repository and click it.



New Blueprint Instance

Click on the repository you'd like to use. If you don't see it below, connect your GitHub or GitLab repository to Render.
Alternatively, to use an existing public repository simply enter the URL below.

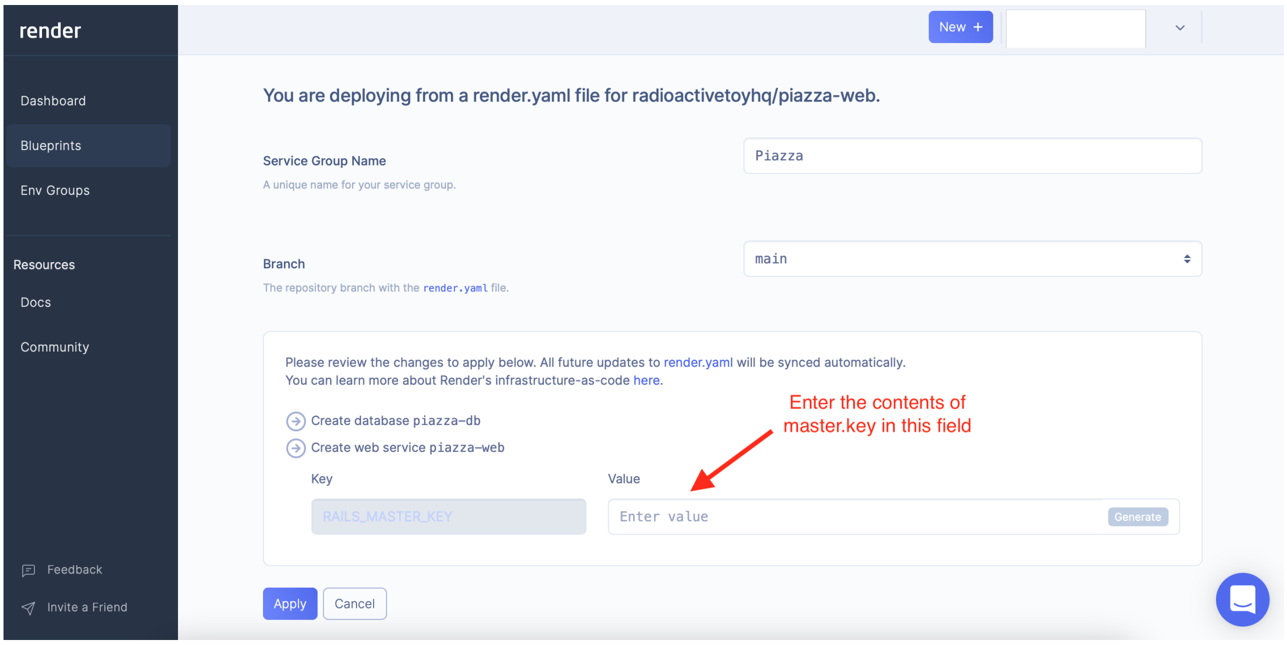
piazza

/ piazza-web Rails app for the Multi platform app development in Rails ...

Not seeing the repository you're looking for? Click here to troubleshoot your GitHub connection.

Figure 5. Connect the piazza repository to Render

This will take you to the page in Figure 6. Enter a name for your service group, I've put in **Piazza**. You'll also need to set the `RAILS_MASTER_KEY` environment variable to decrypt secure data on the server. Open `config/master.key` in your local Rails project and copy its contents to field in the Render dashboard. Then click *Apply*.



You are deploying from a render.yaml file for radioactivetoyhq/piazza-web.

Service Group Name
A unique name for your service group.
Piazza

Branch
The repository branch with the `render.yaml` file.
main

Please review the changes to apply below. All future updates to render.yaml will be synced automatically.
You can learn more about Render's infrastructure-as-code here.

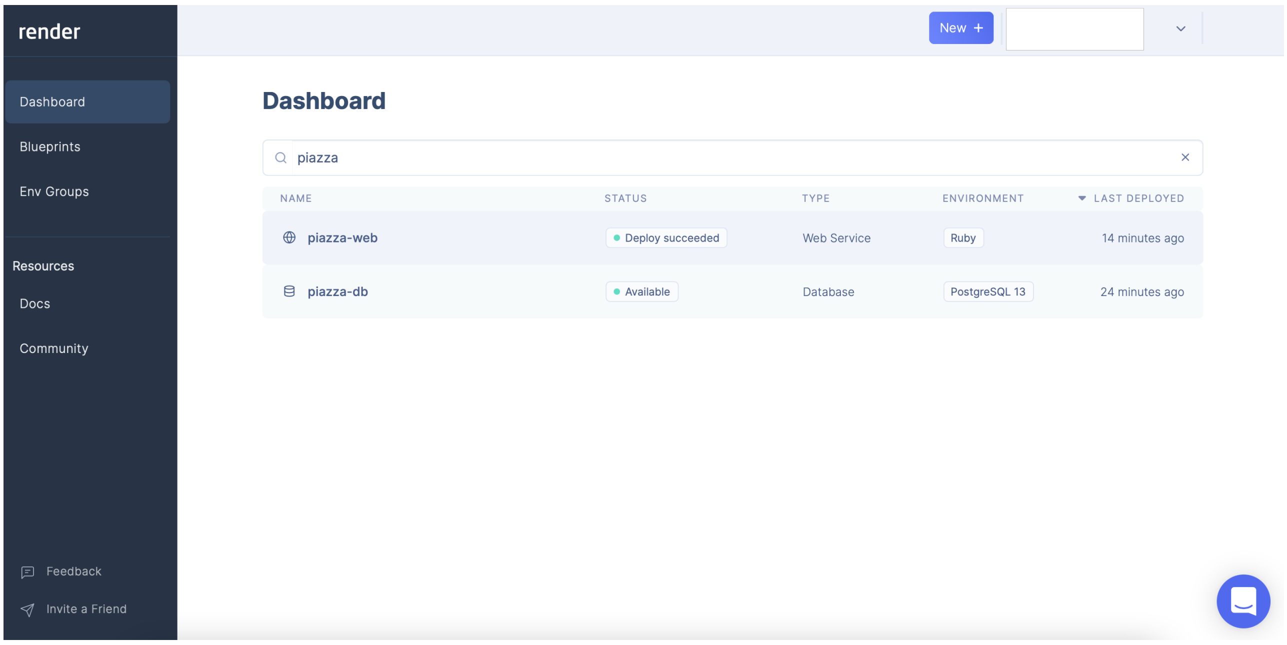
Create database piazza-db
Create web service piazza-web

Key	Value
RAILS_MASTER_KEY	Enter value

Apply Cancel

Figure 6. Deploy the blueprint on Render

Render will create a web service and database. This could take a few minutes to complete. After it's done, you'll be able to see its `Deployed` status in the Render dashboard.



NAME	STATUS	TYPE	ENVIRONMENT	LAST DEPLOYED
piazza-web	Deploy succeeded	Web Service	Ruby	14 minutes ago
piazza-db	Available	Database	PostgreSQL 13	24 minutes ago

Figure 7. The Render dashboard once the services have been deployed

If you click on the **piazza-web** service, it'll take you to the service details page where you'll see a URL for this service as shown in Figure 8. Your URL will be different from the one in the image.

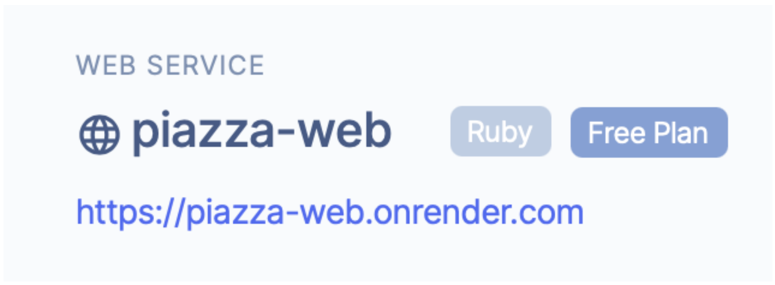


Figure 8. The URL for the web service

Clicking this link will take you to the app, where you'll see an error as we haven't built anything yet. The Rails welcome page is not visible in production. Every time we push code to the `main` branch, Render will automatically deploy it.

That completes the setup for the Rails app.

1.5. Creating the iOS app

To create the iOS project, you'll need a Mac with Xcode installed. If you don't have it installed, you can download it for free from the Mac App Store.

Open Xcode and select **Create a new Xcode project** as shown in Figure 9.

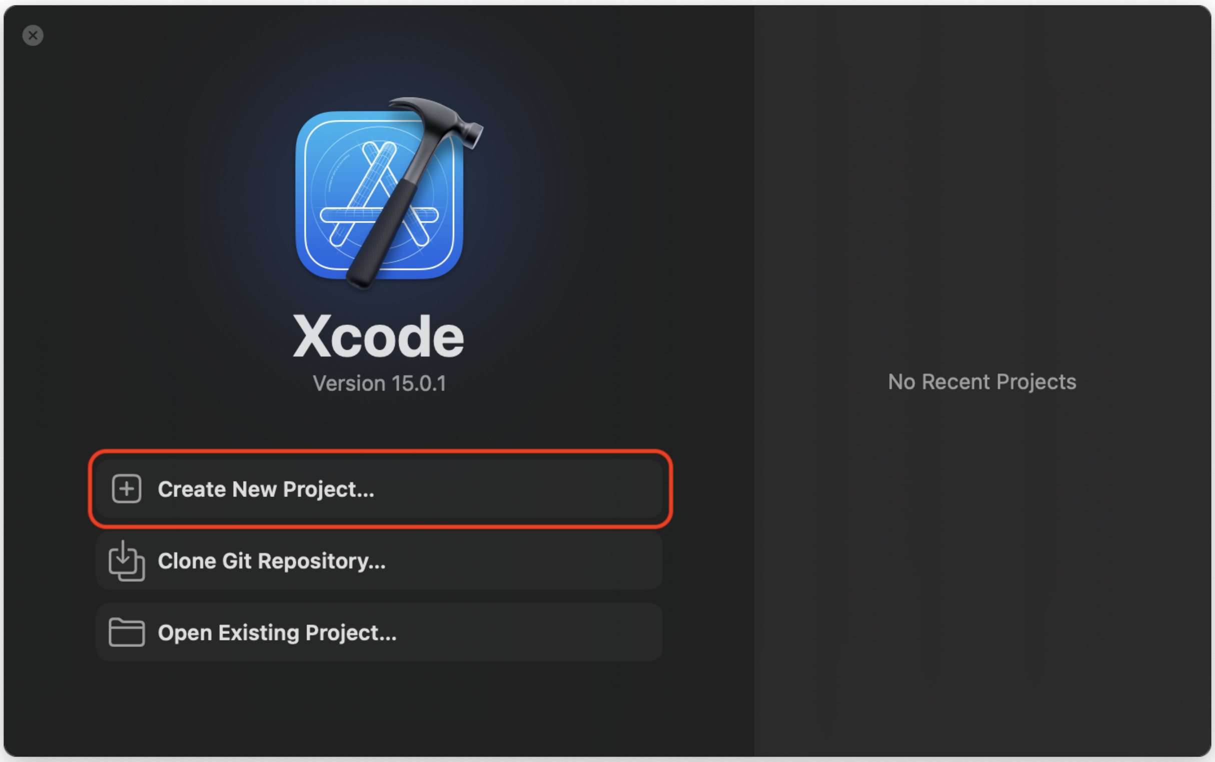


Figure 9. The Xcode splash screen

On the next screen, select **iOS** from the platforms tab and **App** as the template. Click **Next**.

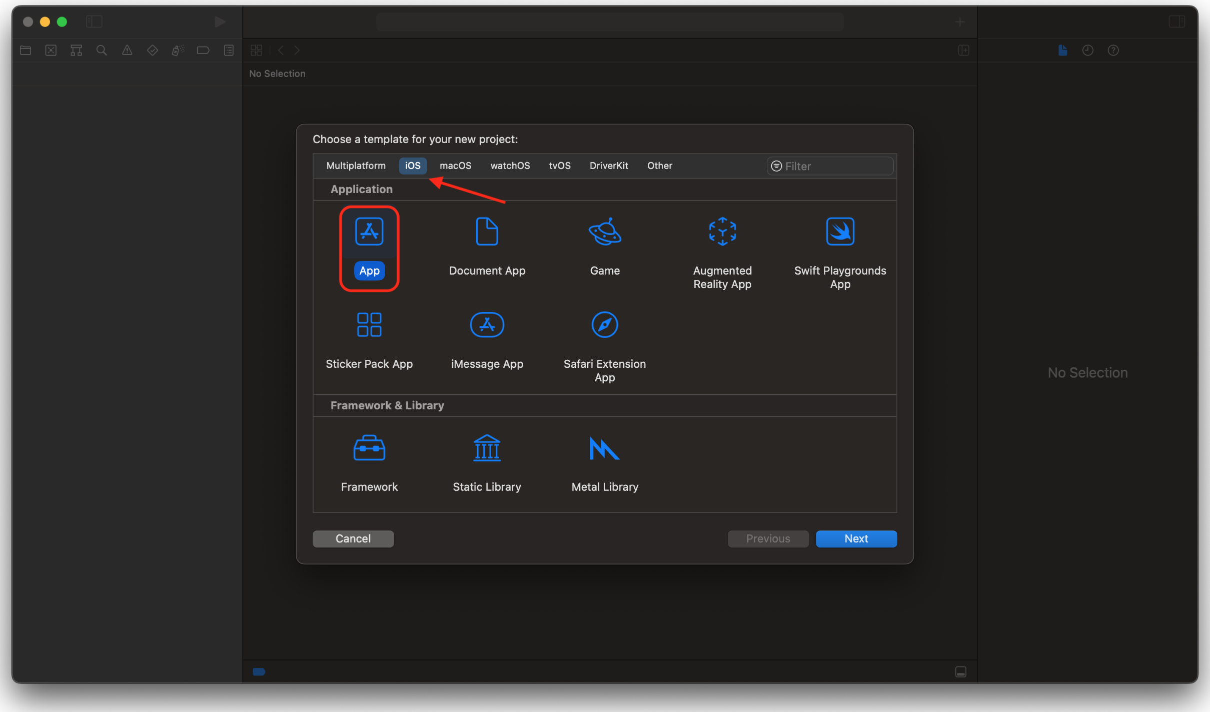
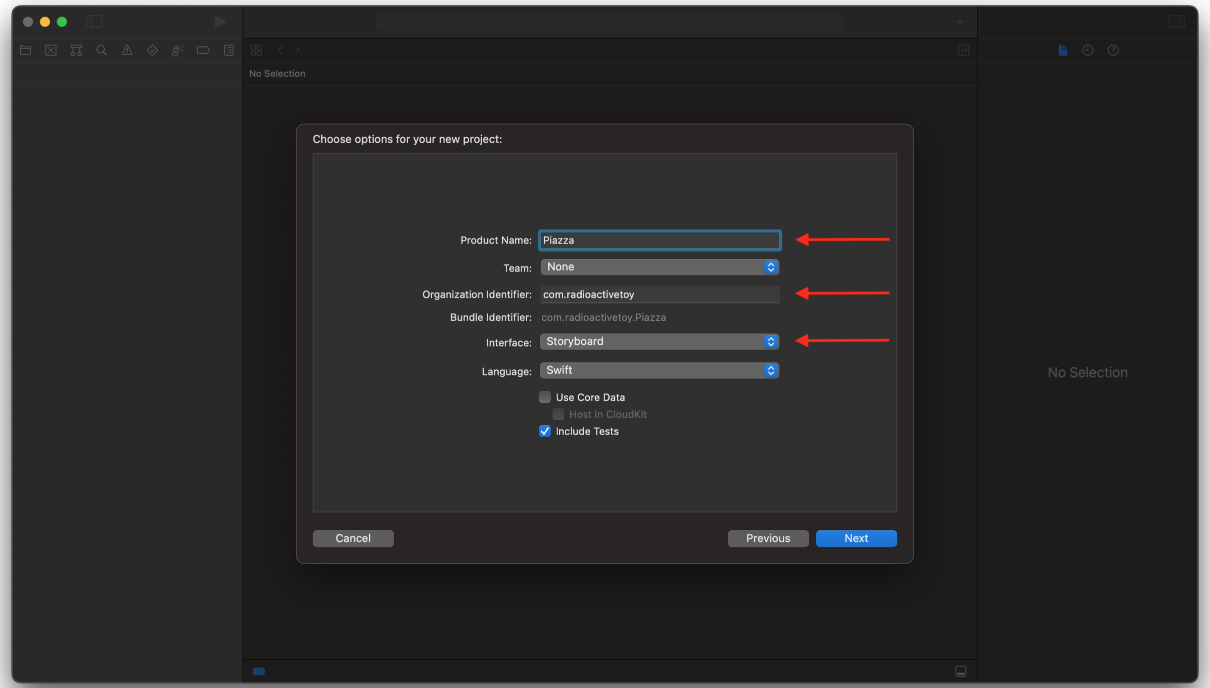


Figure 10. The Xcode template selector

Enter *Piazza* as the **Product Name**. The organization identifier is conventionally the developer's website in reverse DNS notation[5].

I've chosen *com.radioactivetoy* as shown in Figure 11 to denote the publisher of this book. Choose *Storyboard* from the **Interface** dropdown menu. Ensure **Use Core Data** is unchecked and **Include Tests** is checked.


No Selection

Choose options for your new project:

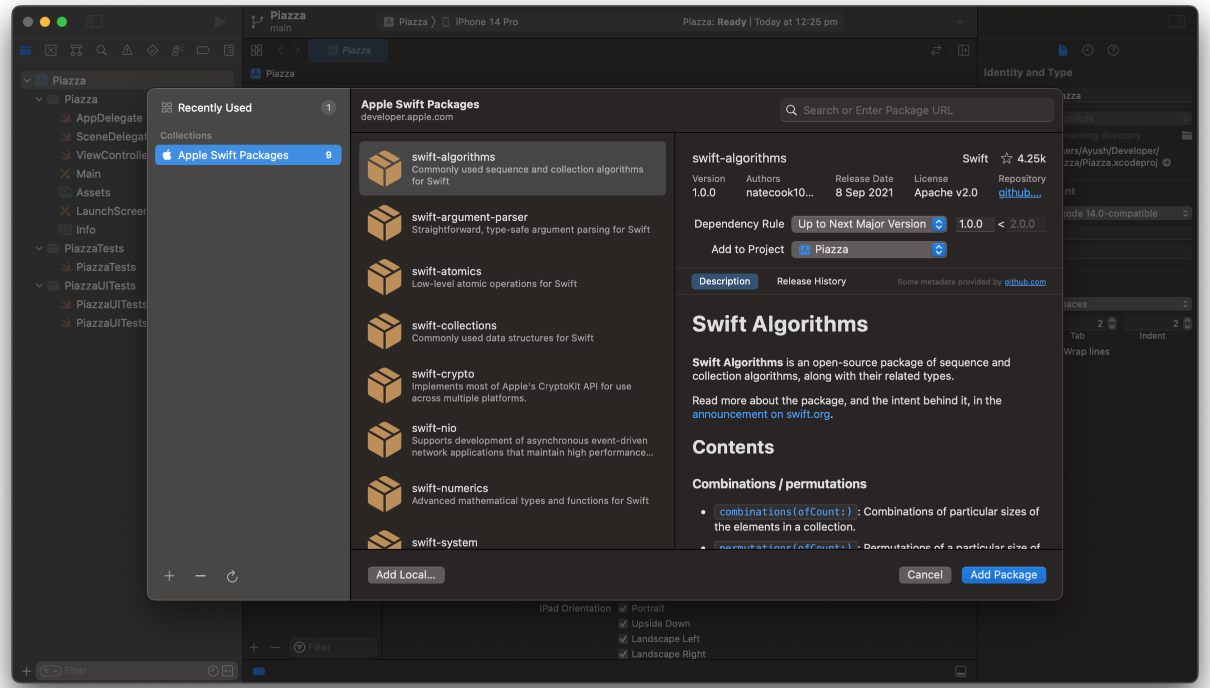
Product Name: Piazza
Team: None
Organization Identifier: com.radioactivetoy
Bundle Identifier: com.radioactivetoy.Piazza
Interface: Storyboard
Language: Swift
Use Core Data
Host in CloudKit
✓ Include Tests

Figure 11. The iOS project details

Click **Next** and select the location for your iOS project. Place it in the same folder as the Rails app. After your project's been created, rename its containing folder from `piazza` to `piazza-ios`.

Now that the iOS project is set up, we need to add the Turbo and Strada iOS packages to the project which area Hotwire's native libraries. We'll get into the details of what these do in the next chapter.

We'll use the native Swift Package Manager to manage dependencies. Select **File → Add Packages** and you should see the dialog box in Figure 12.



Apple Swift Packages
developer.apple.com

swift-algorithms
Commonly used sequence and collection algorithms for Swift

swift-argument-parser
Straightforward, type-safe argument parsing for Swift

swift-atomics
Low-level atomic operations for Swift

swift-collections
Commonly used data structures for Swift

swift-crypto
Implements most of Apple's CryptoKit API for use across multiple platforms.

swift-nio
Supports development of asynchronous event-driven network applications that maintain high performance...

swift-numerics
Advanced mathematical types and functions for Swift

swift-system

swift-algorithms — Swift ☆ 4.25k

Version	Authors	Release Date	License	Repository
1.0.0	natecook10...	8 Sep 2021	Apache v2.0	github....

Dependency Rule: Up to Next Major Version — 1.0.0 < 2.0.0
Add to Project: Piazza

Some metadata provided by github.com

Swift Algorithms

Swift Algorithms is an open-source package of sequence and collection algorithms, along with their related types.

Read more about the package, and the intent behind it, in the announcement on swift.org.

Contents

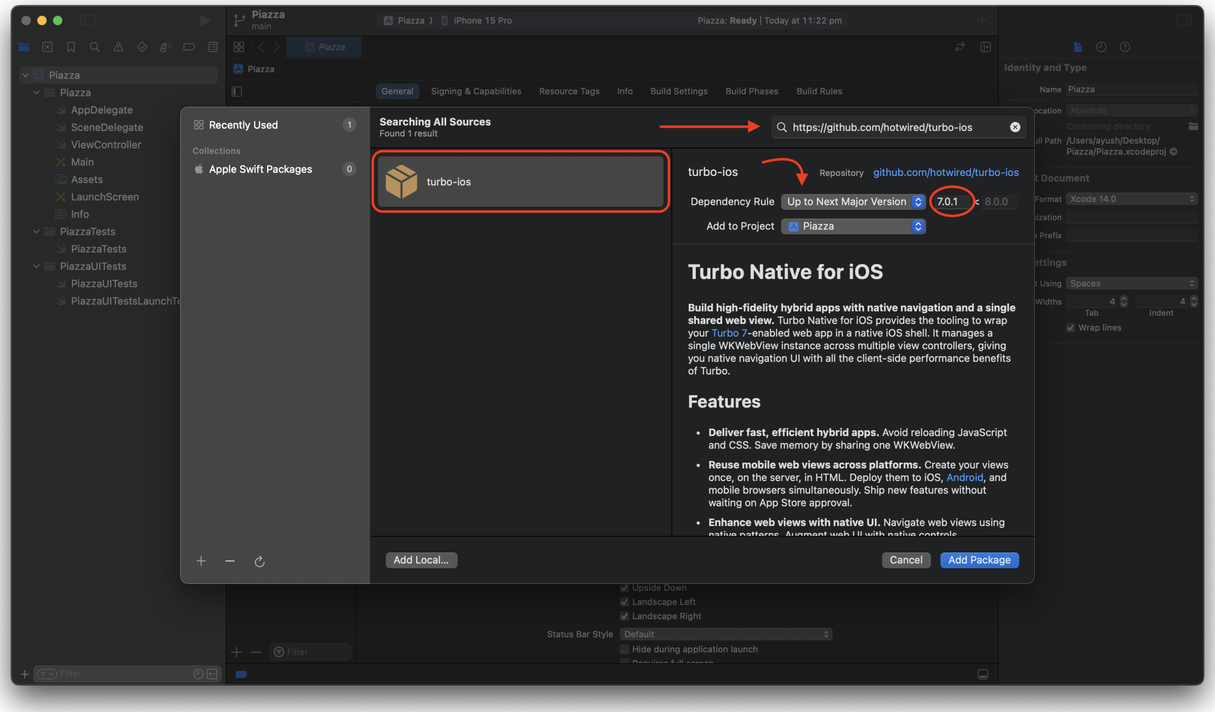
Combinations / permutations

- `combinations(ofCount:)`: Combinations of particular sizes of the elements in a collection.

Figure 12. The dialog box to add Swift packages

In the text field in the top right corner, enter:
`https://github.com/hotwired/turbo-ios`

Select `turbo-ios` from the list of packages. Set the **Dependency Rule** to **Up To Next Minor Version** and enter `7.0.1` in the left text field as demonstrated in Figure 13.


Searching All Sources
Found 1 result

https://github.com/hotwired/turbo-ios

turbo-ios

Repository github.com/hotwired/turbo-ios

Dependency Rule: Up to Next Major Version 7.0.1 < 8.0.0

Add to Project: Piazza

Turbo Native for iOS

Build high-fidelity hybrid apps with native navigation and a single shared web view. Turbo Native for iOS provides the tooling to wrap your Turbo 7-enabled web app in a native iOS shell. It manages a single WKWebView instance across multiple view controllers, giving you native navigation UI with all the client-side performance benefits of Turbo.

Features

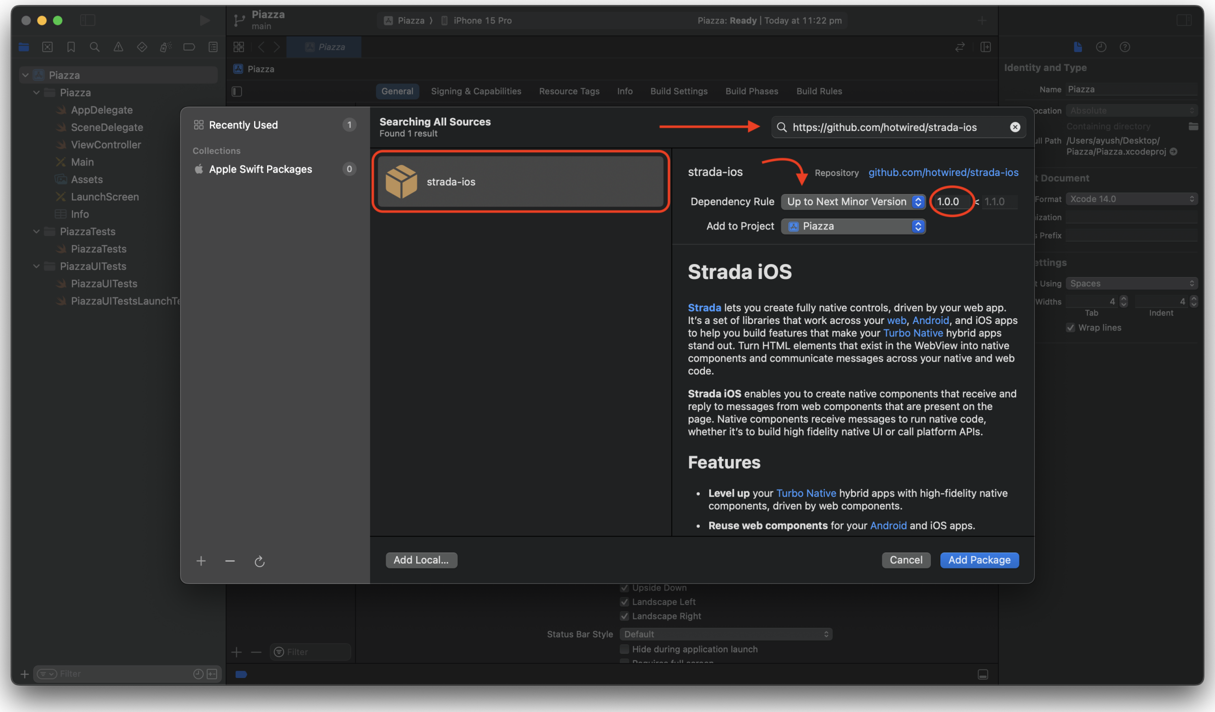
- **Deliver fast, efficient hybrid apps.** Avoid reloading JavaScript and CSS. Save memory by sharing one WKWebView.
- **Reuse mobile web views across platforms.** Create your views once, on the server, in HTML. Deploy them to iOS, Android, and mobile browsers simultaneously. Ship new features without waiting on App Store approval.
- **Enhance web views with native UI.** Navigate web views using

Figure 13. Add the turbo-ios package to the project

Click **Add Package** and you'll see a confirmation screen. Click **Add Package** again.

Open the **Add packages** dialog once again to add Strada. In the text field in the top right corner, enter: https://github.com/hotwired/strada-ios

Select `strada-ios` from the list of packages. Set the **Dependency Rule** to **Up To Next Minor Version** and enter `1.0.0-beta2` in the left text field.



strada-ios

Repository github.com/hotwired/strada-ios

Dependency Rule: Up to Next Minor Version 1.0.0 < 1.1.0

Add to Project: Piazza

Strada iOS

Strada lets you create fully native controls, driven by your web app. It's a set of libraries that work across your web, Android, and iOS apps to help you build features that make your Turbo Native hybrid apps stand out. Turn HTML elements that exist in the WebView into native components and communicate messages across your native and web code.

Strada iOS enables you to create native components that receive and reply to messages from web components that are present on the page. Native components receive messages to run native code, whether it's to build high fidelity native UI or call platform APIs.

Features

- **Level up** your Turbo Native hybrid apps with high-fidelity native components, driven by web components.
- **Reuse web components** for your Android and iOS apps.

Figure 14. Add the strada-ios package to the project

NOTE `1.0.0-beta2` is the latest version at the time of writing. I recommend checking the Releases page for the latest current version and entering that.

This completes the initial setup of the iOS project! It's worth committing our code at this point.

```
$ git add .
$ git commit -m "Complete initial setup of Hotwire"
```

Set up a GitHub repository for this project as well and push your code to it.

1.6. Creating the Android app

Developing Android apps requires Android Studio. If you haven't got it installed, you can download it for free from: https://developer.android.com/studio.

Open Android Studio and select **New Project**.

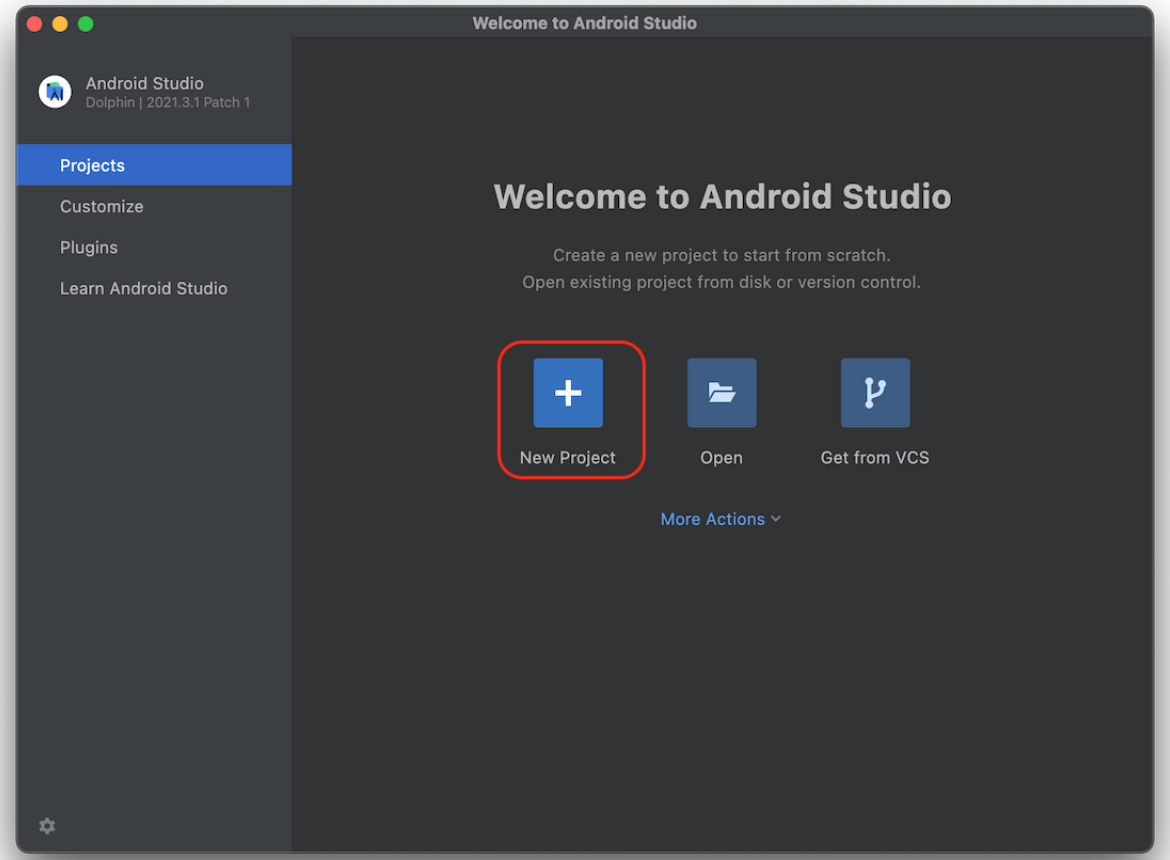

Welcome to Android Studio
Android Studio
Dolphin | 2021.3.1 Patch 1
Projects
Customize
Plugins
Learn Android Studio
Welcome to Android Studio
Create a new project to start from scratch.
Open existing project from disk or version control.
New Project
Open
Get from VCS
More Actions

Figure 15. Create a new Android app project

Choose the **Empty Activity** template from the **Phone and Tablet** category in the template selector. Click **Next**.

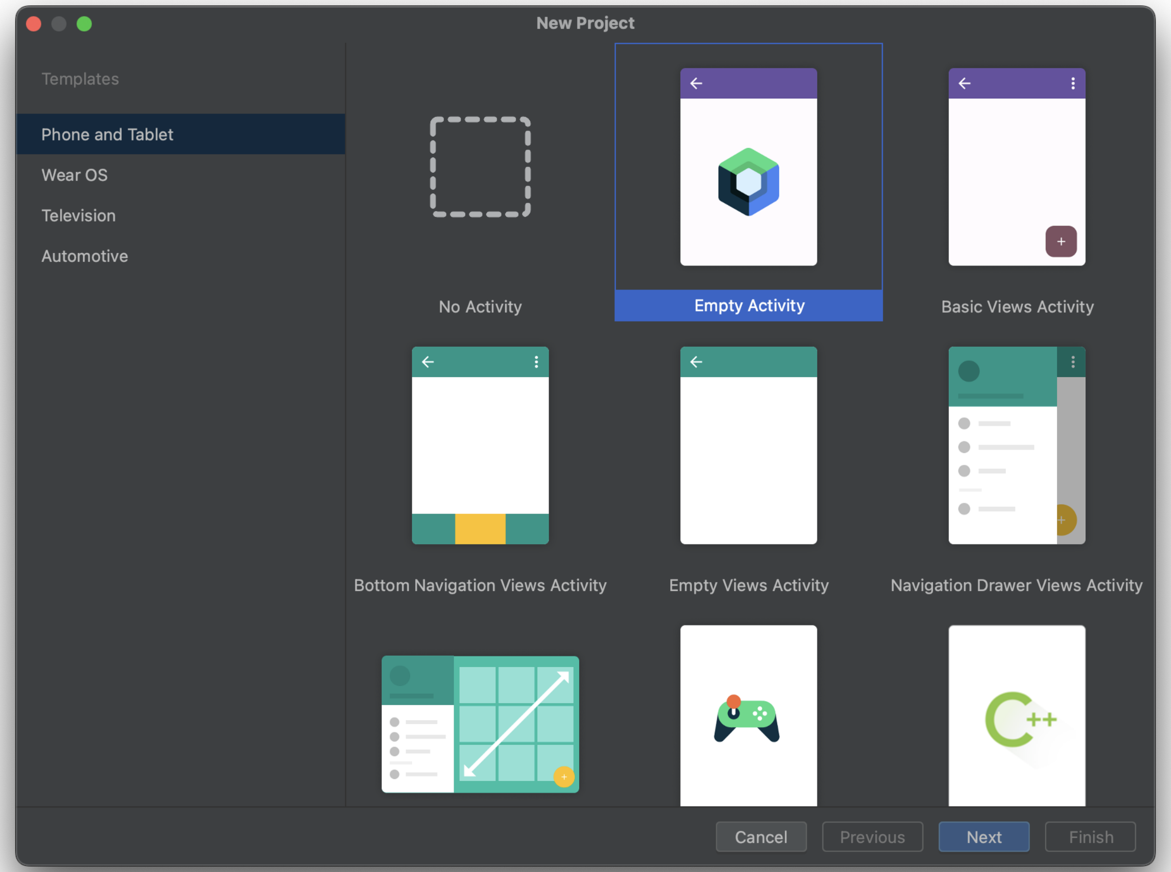
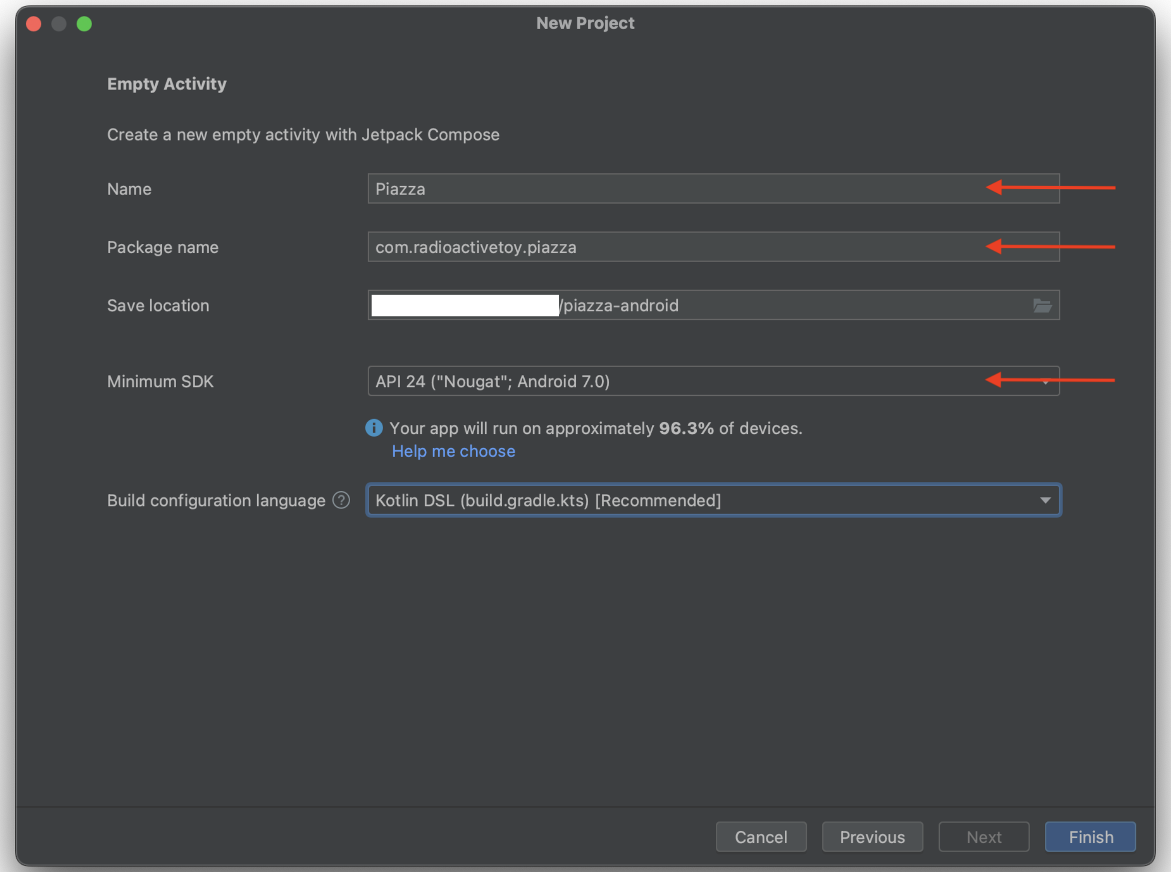


Figure 16. The Android Studio template selector

On the project details page shown in Figure 17, enter *Piazza* for **Name**. Enter a unique package name in the form of a reverse URL. I've entered *com.radioactivetoy.piazza* to denote the publisher of this book.

Create a folder called `piazza-android` alongside your `piazza-web` and `piazza-ios` folders and save the project there.

Select **API 24: Android 7.0 (Nougat)** for the **Minimum SDK**. This is required by Turbo and Strada, which are Hotwire's native packages.



Empty Activity

Create a new empty activity with Jetpack Compose

Name: Piazza

Package name: com.radioactivetoy.piazza

Save location: [redacted]/piazza-android

Minimum SDK: API 24 ("Nougat"; Android 7.0)

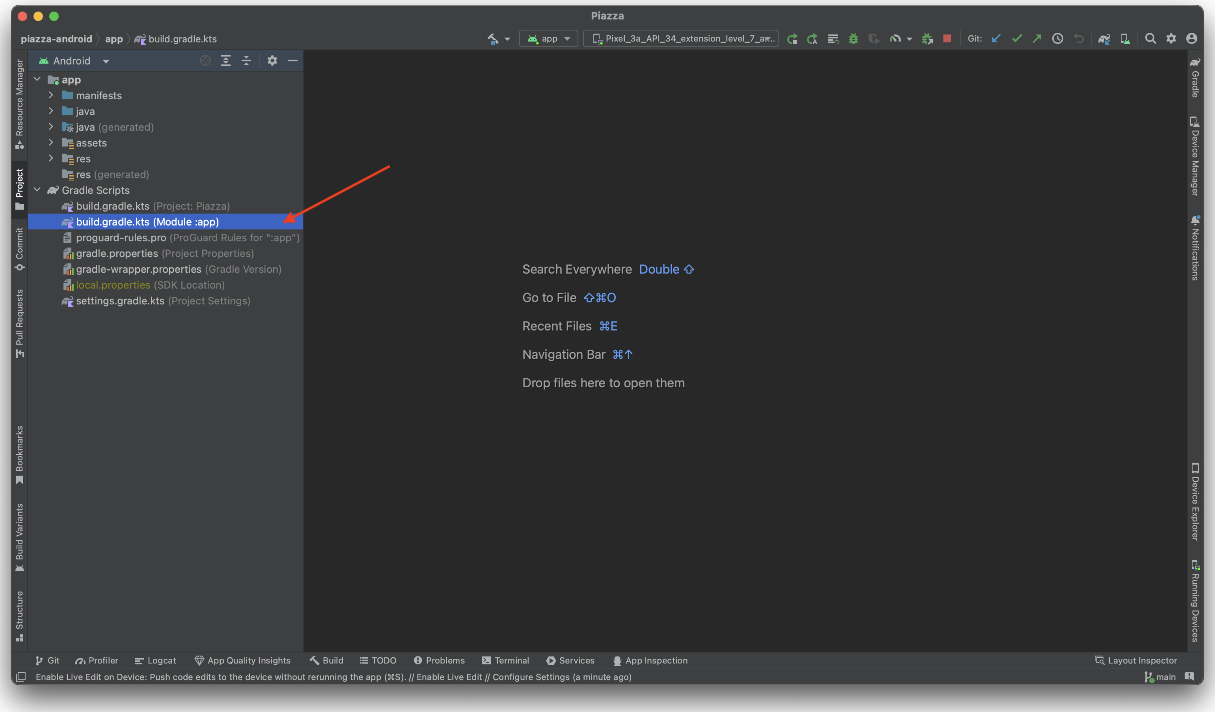
Your app will run on approximately **96.3%** of devices.
Help me choose

Build configuration language: Kotlin DSL (build.gradle.kts) [Recommended]

Figure 17. Enter the details for the Android project

Click **Finish** once you've entered the above information and your project will be created.

The last step is to add the Turbo and Strada Android libraries, along with some other compatibility dependencies to the project. Open the `build.gradle.kts` file for the app's module as shown in Figure 18.


Piazza
piazza-android › app › build.gradle.kts
app
Pixel_3a_API_34_extension_level_7_ar...
Git:
Android
app
manifests
java
java (generated)
assets
res
res (generated)
Gradle Scripts
build.gradle.kts (Project: Piazza)
build.gradle.kts (Module :app)
proguard-rules.pro (ProGuard Rules for ":app")
gradle.properties (Project Properties)
gradle-wrapper.properties (Gradle Version)
local.properties (SDK Location)
settings.gradle.kts (Project Settings)
Search Everywhere Double ⇧
Go to File ⇧⌘O
Recent Files ⌘E
Navigation Bar ⌘↑
Drop files here to open them
Resource Manager
Project
Commit
Pull Requests
Bookmarks
Build Variants
Structure
Gradle
Device Manager
Notifications
Device Explorer
Running Devices
Git
Profiler
Logcat
App Quality Insights
Build
TODO
Problems
Terminal
Services
App Inspection
Layout Inspector
main

Enable Live Edit on Device: Push code edits to the device without rerunning the app (⌘S). // Enable Live Edit // Configure Settings (a minute ago)

Figure 18. The build.gradle.kts file for our app

Scroll down to the `dependencies` declaration and add the library as demonstrated in Listing 6.

Listing 6. Add the required packages to Piazza (`app/build.gradle.kts`)

```
plugins {
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
    kotlin("plugin.serialization") version "1.8.10"
}

// ...

dependencies {
  implementation("androidx.core:core-ktx:1.9.0")
  implementation("androidx.appcompat:appcompat:1.5.1")
  implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.6.1")
  implementation("androidx.activity:activity-compose:1.7.0")
  implementation(platform("androidx.compose:compose-bom:2023.03.00"))
  implementation("androidx.compose.ui:ui")
  implementation("androidx.compose.ui:ui-graphics")
  implementation("androidx.compose.ui:ui-tooling-preview")
  implementation("androidx.compose.material3:material3")
  implementation("androidx.browser:browser:1.4.0")
  implementation("dev.hotwire:turbo:7.0.+")
  implementation("dev.hotwire:strada:1.0.0-beta3")
  implementation("org.jetbrains.kotlinx:kotlinx-serialization-
json:1.6.0")

  testImplementation("junit:junit:4.13.2")

  androidTestImplementation("androidx.test.ext:junit:1.1.5")
  androidTestImplementation("androidx.test.espresso:espresso-
core:3.5.1")
  androidTestImplementation(platform("androidx.compose:compose-
bom:2023.03.00"))
  androidTestImplementation("androidx.compose.ui:ui-test-junit4")

  debugImplementation("androidx.compose.ui:ui-tooling")
  debugImplementation("androidx.compose.ui:ui-test-manifest")
}
```

We also need to update the Project level `build.gradle.kts` file to enable a plugin required by Strada.

Listing 7. Enable the serialization plugin (`build.gradle.kts`)

```
// Top-level build file where you can add configuration options
common to all sub-projects/modules.
plugins {
    id("com.android.application") version "8.1.3" apply false
    id("org.jetbrains.kotlin.android") version "1.8.10" apply false
    kotlin("plugin.serialization") version "1.8.10"
}
```

Once you've added those lines, Android Studio will prompt you to *Sync* the project. Go ahead and sync it.

That completes the setup for the Android app! Initialize a `git` repository for the project and commit the code.

```
$ git init
$ git add .
$ git commit -m "Set up android app with Hotwire"
```

Set up a GitHub repository for the Android project and push your code to it.

1.7. Conclusion

Well that was a bit of a slog, but we’re up and running on all 3 platforms now! The Rails app has been created and deployed to Render and the two native app projects have been set up with their Turbo Native extensions.

Next, let’s take a closer look at what Hotwire actually is.

[1] Rails 7 uses importmaps by default to avoid the "build" step for JavaScript. However, bundling JavaScript is still a good idea. This Twitter thread from Konnor Rogers elaborates: https://twitter.com/RogersKonnor/status/1425922887161090055.

[2] A detailed comparison of the benefits of Render in comparison to Heroku can be found here: https://render-web.onrender.com/render-vs-heroku-comparison.

[3] See the limitations of Render’s free tier: https://render.com/docs/free.

[4] Defining our deployment infrastructure as code has the advantage of being tracked in version control along with our code. So all changes are tracked and any new issues introduced will be easier to find. See Render’s complete YAML spec: https://render.com/docs/blueprint-spec.

[5] Some guidance on iOS organization identifier can be found here: https://stackoverflow.com/a/35102666

Chapter 2. What is Hotwire?

Hotwire is a suite of frontend libraries that enable us to build rich, high-fidelity, and modern web applications without using monolithic frontend frameworks like React or Vue. In fact, we'll write very little JavaScript ourselves!

Hotwire also provides native extensions for iOS and Android using which we can build native *shell* apps which reuse the views from the web app.

The libraries that constitute Hotwire are ***Turbo***, ***Stimulus***, and ***Strada***.

Turbo is the nucleus of Hotwire. It speeds up page changes and form submissions. It also allows us to divide complex pages into *frames* and then make partial page updates confined to those *frames*. All this without writing a single line of custom JavaScript!

The core of the native extensions for iOS and Android build on top of Turbo, known as ***Turbo Native***. More on this in the next section!

Stimulus is a lightweight mechanism using which we can attach pieces of JavaScript logic to HTML elements in the form of a *controller*. It's meant for use cases where custom JavaScript is required; for example, to hide or show a particular element on the click of a button.

Strada augments Turbo Native by bridging native code with the web app using JavaScript. It standardises an interface to communicate between the web app and the native app allowing us to render fully native components that act as a proxy for web elements.

Let's dissect each one of Hotwire's constituent libraries.

2.1. Turbo

Turbo is the successor to Turbolinks which was included in Rails prior to v7.

It was initially written in ***TypeScript*** and then refactored to ***JavaScript*** for v8.

Turbo itself is made up of 4 parts: ***Turbo Drive***, ***Turbo Frames***, ***Turbo Streams***, and ***Turbo Native***.

2.1.1. Turbo Drive

Turbo Drive speeds up form submissions and navigation between pages by eliminating the need for full-page reloads. One of the main performance bottlenecks when loading a webpage is downloading and parsing the JavaScript and CSS. This happens for every single visit to every single page in an application even though JavaScript and CSS bundles seldom change. Imagine the performance win if these bundles could be downloaded and parsed only once! That's exactly what Turbo Drive enables.

It intercepts every link click and form submission and performs the HTTP request in the background using `fetch`. It then replaces the content of the `<body>` tag on the current page with that of the new page loaded in the background. The new page's `<head>` is merged in with the existing `<head>`.

The browser's URL bar is updated with the new location ensuring the functionality of the *Back* button is preserved.

2.1.2. Turbo Frames

A ***Turbo Frame*** is a part of a web page that can be updated in isolation from the rest of the web page. All link clicks and form submissions made from within a Turbo Frame will result in only that particular Frame being updated with the response, rather than the full page. Using frames, a complex page can be decomposed into independent parts.

Let's take a look at some code to see how this works. A Turbo Frame is declared using the `<turbo-frame>` tag with a unique ID. The below snippet demonstrates a Turbo Frame containing a list with a *refresh* link.

```
<body>
```

```html
  <!-- ... -->

  <turbo-frame id="comments">
    <ul>
      <li>A comment</li>
      <li>Another comment</li>
    </ul>

    <a href="/comments">Refresh</a>
  </turbo-frame>

  <!-- ... -->
</body>
```

When the *refresh* link is clicked, Turbo will make a `GET` request to `/comments`. Since the navigation happened from inside a Turbo Frame, it will expect the response to contain a `<turbo-frame>` with the ID `comments`. Turbo will then update the Frame with the contents of the corresponding Frame returned in the response, leaving the rest of the page untouched.

Links inside a Frame can be designated to behave as normal links and trigger a full page visit. This is done using the `data-turbo-frame` attribute.

```html
<body>
  <!-- ... -->

  <turbo-frame id="modal">
    <p>
      <!-- ... -->
    </p>

    <!-- This link will trigger a full page visit -->
    <a href="/home" data-turbo-frame="_top">Go back</a>
  </turbo-frame>

  <!-- ... -->
```

```
</body>
```

Conversely, links outside a Frame can be designated to behave as if they were within a Frame using the same `data-turbo-frame` attribute.

```
<body>
  <!-- ... -->

  <turbo-frame id="comments">
    <ul>
      <li>A comment</li>
      <li>Another comment</li>
    </ul>
  </turbo-frame>

  <!-- This link will only update the above Turbo Frame -->
  <a href="/comments" data-turbo-frame="comments">Refresh</a>
  <!-- ... -->
</body>
```

Turbo Frames can also be loaded remotely.

```
<body>
  <!-- ... -->

  <turbo-frame id="comments" src="/comments">
  </turbo-frame>
</body>
```

The above Frame will be loaded from `/comments` as soon as it's added to the DOM. The response will still need to contain a `<turbo-frame>` with a matching ID.

To enhance performance, a `loading="lazy"` option can be defined on remotely loaded Frames. This way, the Frame won't be loaded until it's visible to the user.

```html
<body>
  <!-- ... -->

  <turbo-frame id="comments" src="/comments" loading="lazy">
  </turbo-frame>
</body>
```

Using Turbo Frames, we can update specific self-contained parts of the page with ease and without writing any custom JavaScript!

2.1.3. Turbo Streams

Turbo Streams enable us to perform a series of CRUD actions on specific DOM elements via a `<turbo-stream>` tag.

As soon as a `<turbo-stream>` tag is added to the document, Turbo will execute it and perform the alteration it defines. These tags can be delivered in a variety of methods such as an HTTP response, WebSocket, or Server Sent Events. We'll cover some of these methods while building Piazza.

A typical Turbo Stream tag looks like:

```html
<turbo-stream action="append" target="comments">
  <template>
    <li id="comment_1">
      This li will be appended to the element
      with the DOM ID "comments".
    </li>
  </template>
</turbo-stream>
```

As the above example illustrates, a `<turbo-stream>` must have an `action` and `target` attribute. The `action` will be performed on the element with the ID defined in `target`. It must contain a `<template>` tag which defines the contents of update to be made.

Listing 8 demonstrates the `action` types supported by Turbo Streams.

Listing 8. All supported Turbo Streams actions

```
<turbo-stream action="append" target="comments">
  <template>
    <li id="comment_1">
      This li will be appended to the element
      with the DOM ID "comments".
    </li>
  </template>
</turbo-stream>

<turbo-stream action="prepend" target="comments">
  <template>
    <div>
      This div will be prepended to the element
      with the DOM ID "comments".
    </div>
  </template>
</turbo-stream>

<turbo-stream action="replace" target="comment_1">
  <template>
    <li id="comment_1">
      This li will replace the existing element
      with the DOM ID "comment_1".
    </li>
  </template>
</turbo-stream>

<turbo-stream action="update" target="unread_count">
  <template>
    <!-- The contents of this template will replace the
    contents of the element with ID "unread_count" by
    setting innerHtml to "" and then switching in the
    template contents. -->
    1
  </template>
</turbo-stream>
```

```
<turbo-stream action="remove" target="comment_1">
  <!-- The element with DOM ID "comment_1" will be removed.
  The contents of this stream element are ignored. -->
</turbo-stream>

<turbo-stream action="before" target="current_step">
  <template>
    <!-- The contents of this template will be added before the
    the element with ID "current_step". -->
    <li>New item</li>
  </template>
</turbo-stream>

<turbo-stream action="after" target="current_step">
  <template>
    <!-- The contents of this template will be added after the
    the element with ID "current_step". -->
    <li>New item</li>
  </template>
</turbo-stream>
```

Multiple elements can be targeted using a CSS selector defined in the `targets` attribute.

```
<turbo-stream action="append" targets="li.unread">
  <template>
    <div>
      This div will be appended to all
      li elements with the "unread" class
    </div>
  </template>
</turbo-stream>
```

Another great thing about Streams is that the method by which a `<turbo-stream>` is added to the document doesn't matter. For example, if we wanted to update a Turbo Frame along with one specific element that's outside the Frame, a `<turbo-stream>` can be rendered within the Turbo

Frame returned by the server. When the Frame is updated, the stream will be executed!

Custom Stream actions

In addition to the stock Turbo Stream actions, we can define custom actions to do whatever we want. Let's look at an example to add a CSS class to an element.

```
import { StreamActions } from "@hotwired/turbo"

StreamActions.add_class = function() {
  let className = this.getAttribute("class")

  this.targetElements.forEach(e => {
    e.classList.add(className)
  })
}
```

In custom actions, `this` refers to `StreamElement`, which is the custom element underpinning `<turbo-stream>`. The `targetElements` getter is defined by this element.

The above action can be triggered as:

```
<turbo-stream action="add_class"
              targets="li.unread"
              class="is-unread">
```

Turbo Streams gives us a scalpel to make fine grained amendments to the page as opposed to the more general approach of Turbo Drive and Frames.

2.1.4. Page refreshes using morphing

A common pattern in Rails is to redirect the user back to the page they were on after a form submission. This usually happens after the user edits or creates an entity, with the changes reflected in the UI after the redirect.

Redirecting triggers a full page load which resets the user's scroll position and the state of the entire page. In cases where this isn't desirable, we add some complexity on the server and render Turbo Streams to make localised changes to the page.

Page refreshes using morphing is an alternative approach to this problem which eliminates the additional complexity on the server. When using this approach, the server redirects as it usually would after form submission. If the redirect is back to the page's current location, Turbo will fetch the page again and *morph* the new page with the current one, meaning that only the elements which have changed will be updated. The scroll position will also be preserved.

Under the hood, Turbo uses `idiomorph` to execute the morphing[1].

Let's see what this looks like in action.

A demo

Let's say we have an index of *posts* on a page, each with a **Delete** button.

```
<!-- /posts -->
<h2>You have <strong>21 posts<strong></h2>

<ol>
  <li>
    <h3>My favourite albums of 2023</h3>
    <form method="delete" action="/post/1">
      <button type="submit">Delete</button>
    </form>
  </li>

  <li>
    <h3>Enter The Harmony Codex</h3>
    <form method="delete" action="/post/2">
      <button type="submit">Delete</button>
```

```html
    </form>
  </li>

  <!-- ... -->
</ol>
```

When the user deletes a post, reloading the full page would reset their scroll position and feel jarring. Alternatively, we could render Turbo Streams from the server to remove just that post's `<li>` element and update the counter. This would add complexity as we'd need to set a DOM ID on each `<li>` element and render a Turbo Stream template on the server to remove the appropriate element and change the counter.

The complexity would increase dramatically if we had other bits of state scattered around the page. This is the perfect use case for morphing. First we need to opt into it using a `<meta>` tag.

```html
<head>
  <!-- ... -->

  <!-- The refresh method can be `morph` or `replace` -->
  <meta name="turbo-refresh-method" content="morph">

  <!-- The scroll setting can be `preserve` or `reset` -->
  <meta name="turbo-refresh-scroll" content="preserve">
</head>
```

These are page-specific settings. Having two definitions allows us to control the refresh method and scroll setting on refresh independently.

Now, on the server, we can simply redirect the user after deleting the post.

```ruby
class PostsController < ApplicationController
  # ...

  def delete
    @post.destroy
    redirect_to posts_path
  end
end
```

Since we've opted into morphing on this page, and the redirect is back to the current location, Turbo will use idiomorph to update the page. Hence, only the counter will be updated and the appropriate post's `<li>` will be removed.

The scroll position will also be retained making the user's experience more slick.

Turbo also provides a Stream Action to trigger a page refresh.

```
<turbo-stream action="refresh"></turbo-stream>
```

Morphing only works for full-page refreshes. It isn't scoped to Turbo Frames. Those will always be fully reloaded with one exception.

Turbo Frames and morphing

When a page that has opted into morphing contains remotely loaded Turbo Frames (those with an `src` attribute), we can opt into morphing for those Frames.

By default, they'll be reloaded on a page refresh without morphing, but we can reload them with morphing using `refresh="morph"`.

```
<turbo-frame id="comments" src="/comments" refresh="morph">
</turbo-frame>
```

When the page is reloaded, this Frame will also be reloaded using morphing.

Inline Turbo Frames (without an `src` attribute) will be morphed like any other element. Navigation within Turbo Frames doesn't currently support morphing so those will be reloaded as normal.

Put together, Turbo as a whole packs one hell of a punch and we can accomplish an inordinate number of use cases without writing a single line of JavaScript!

2.1.5. Turbo Native

Turbo Native consists of native libraries for iOS and Android that harness the power of Turbo on mobile. It orchestrates a native *web view* through different screens within native navigation. This way, native components such as *tab bars* can be used, but the same HTML views used on web are leveraged to display content in the app.

Tapping a link in the app triggers a new native screen to be displayed with the destination page giving the app a distinct native feel. The page navigation is driven using Turbo and same *web view* instance is reused across the different native screens so the performance is incredibly slick.

For cases where a web view doesn't quite cut it, Turbo Native has an escape hatch to display a fully native screen instead. We'll dig deeper into this in chapters ahead as it's easier to understand when actually writing the code!

The Turbo documentation is available at https://turbo.hotwired.dev. We'll cover most of the functionality over the course of this book, but it's worth familiarising yourself with it. Documentation is your best friend!

2.2. Stimulus

Stimulus is a lightweight library to manipulate HTML with reusable pieces of JavaScript logic encapsulated in a *controller*. Like Turbo, it's written in ***TypeScript***.

Stimulus has an HTML-centric way of writing JavaScript. The markup is connected to the controller using a range of `data-` attributes. Stimulus controllers should ideally be reusable and generic.

Let's look at a concrete use case. Say the visibility of an element needed to be toggled on the click of a button. The HTML for this would be something like:

```html
<article>
  <div>
    This is a thing.
  </div>

  <button>Toggle the thing!</button>
</article>
```

When the button is clicked, we want to hide or show the `<div>`. This entails adding or removing a `hidden` class on the `<div>` which will hide it using CSS. The above markup decorated with `data-` attributes to hook it up to a Stimulus controller looks like:

```html
<article data-controller="toggle">
  <div data-toggle-target="item">
    This is a thing.
  </div>

  <button
    data-action="click->toggle#toggleItem"
    data-toggle-class-value="hidden">
    Toggle the thing!
  </button>
</article>
```

The `<article>` is connected to a controller called `toggle` using `data-controller`. This means the `toggle` controller's scope is `<article>` and it will only be able to "see" this element and what's inside it.

The `<div>` whose class needs to be toggled is denoted as a *target* named `item` using the `data-toggle-target` attribute. A *target* is an element that needs to be operated upon in a Stimulus controller. It can be accessed using `this.[target name]Target`. The pattern for a target attribute is:

```
data-[controller name]-target
```

Multiple elements in the controller's scope can be decorated with the same *target* name. They can all be accessed using `this.[target name]Targets`.

On the `<button>`, we specify the controller method to trigger for a specified event using `data-action`. In this case it's triggering the `toggleItem` method in the `toggle` controller on a `click` event. For `<button>`, the default action event[2] is `click` so that can be omitted and the attribute value becomes: `toggle#toggleItem`. But for demonstration purposes, let's be explicit. A `data-action` definition is of the form:

```
[event]->[controller name]#[method]
```

We're also defining a *value* called `class` using `data-toggle-class-value`. A *value* defines parameters that can be accessed within a Stimulus controller using `this.[value name]Value`. Defining the CSS class to be toggled as a value allows the controller to be generic and reusable in a variety of contexts, not just to show or hide an element. A value attribute is defined as:

```
data-[controller name]-[value name]-value
```

Values should be defined on the same element as `data-controller`.

Next, let's write the controller. Rails creates an `app/javascript/controllers` folder by default. It also creates files that initialize Stimulus and register all the controllers. We'll get into how all that works when we write our first controller in Piazza. For now, let's just say the controller goes in a file called `toggle_controller.js` and that file needs to be registered with the Stimulus *application* as below:

```
import { Application } from "@hotwired/stimulus"
const application = Application.start()

import ToggleController from "./toggle_controller"
application.register("toggle", ToggleController)
```

`application.register` defines controller's name as `toggle`, which is what we use in the HTML. The controller's implementation is demonstrated below:

```
import { Controller } from "stimulus"

export default class extends Controller {

  static targets = [ "item" ]
  static values = { class: String }

  toggle() {
    this.itemTarget.classList.toggle(this.classValue)
  }
}
```

Let's unpack it to understand what it's doing. We've defined a `toggle()` method which is triggered by the `<button>` as defined in `data-action`.

```
toggle() {
  this.itemTarget.classList.toggle(this.classValue)
}
```

A single `target` called *item* in an array has been declared. This points to the `<div>` decorated with `data-toggle-target="item"` and is available within the controller using `this.itemTarget`.

```
static targets = [ "item" ]
```

We've also declared a `value` called `class` of type `String`. This is accessible using `this.classValue` in the controller and will read or write the `data-toggle-class-value` attribute defined in the HTML.

```
static values = { class: String }
```

In essence, that's Stimulus! It lets you hook up pieces of logic to HTML elements and provides an elegant API for interacting with attributes defined in the DOM. It's conceptually super simple but the complete API is extremely powerful.

Other features include the Outlets API which provides a mechanism to communicate between different Stimulus controllers; and the Classes API makes it easy to work with utility CSS classes. Check out the docs for all the details.

The Stimulus documentation is available at https://stimulus.hotwired.dev. We'll cover the basics in this book but to level up your knowledge, it's totally worth reading the docs!

2.3. Strada

Strada provides a way to send and receive messages from native code encapsulated within Stimulus controllers. It consists of a JavaScript library for the web, and Swift and Kotlin libraries for iOS and Android.

Strada provides a subclass of a Stimulus `Controller` called `BridgeComponent`. This construct is the abstraction layer for message passing. The native apps will have counterpart components written in native code to receive and respond to web messages.

Listing 9 demonstrates a basic component which sends a message to the native app to render a link, and then handles the response when the native component is clicked by clicking the web element.

Listing 9. A basic Strada web component

```
import { BridgeComponent, BridgeElement } from "@hotwired/strada"

export default class extends BridgeComponent {
  static component = "link"

  connect() {
    const title = new BridgeElement(this.element).title

    this.send("connect", { title }, response => {
      console.log(`Response: ${response}`)
      this.element.click()
    })
  }
}
```

The app can render the link in a platform specific way. It's up to the developer to handle the message from the web app in the appropriate way.

Strada also provides a `BridgeElement` helper class which encapsulates and infers basic information such as the element's title from the HTML. It also provides an interface to `data-bridge-` HTML attributes which we can use to define custom information.

We'll dig into the specifics further in Chapter 7 as it's easier to understand when writing web and native code concurrently.

2.4. Conclusion

Over the course of this chapter, we've taken a tour of Hotwire's constituent libraries.

To recap, Turbo speeds up page navigation and form submissions by eliminating the need for full-page reloads. It also enables updates of specific portions of a page using Turbo Frames and small, targeted amendments using Turbo Streams.

Stimulus provides the ability to hook up pieces of JavaScript logic to HTML elements using data attributes, and an elegant API to access the data within these attributes.

Strada standardises an interface to communicate between web and native code allowing us to build hi-fidelity native UIs which are completely driven by the web app.

Hotwire, when put together with a full-stack framework such as Rails packs in the needs of the vast majority of applications and allows individuals or small teams to develop apps at an incredible pace. Thanks to Hotwire, it’s possible for a single developer to build and maintain an app for Web, iOS, and Android without tying themselves into knots!

[1] Radan Skorić has written a great walkthrough of the Idiomorph algorithm: https://radanskoric.com/articles/turbo-morphing-deep-dive-idiomorph.

[2] The full list of default events for the various elements can be found here: https://stimulus.hotwired.dev/reference/actions#event-shorthand.

Chapter 3. Users and sign ups

Now that we've set up our repositories and had a quick overview of Hotwire, let's get started with building Piazza!

What's a web app without any users? We'll start by building a *sign up* form enabling new users to create an account on Piazza. This will be followed up by building an *authentication* system so users can log in and the application can securely identify them.

The first step is creating the `User` and `Organization` models.

3.1. Modelling Users and Organizations

`User`s form one half of the backbone of our app. The other half is the oft overlooked `Organization`. A `User` belongs to an `Organization` and all domain resources belong to the `Organization` instead of the individual `User`.

More often than not, the concept of *organizations* is considered to be out of scope for MVPs[1], usually with good reason. Management of organization members, the designation of roles and permissions, and an invitation system to join an organization are all essential and non-trivial features.

In the context of the domain model however, it's critical to have the concept of an `Organization`. Otherwise, all the resources and permissions would be built around an individual `User`.

Why use the term "Organization"?

A variety of terms such as `Team`, `Company`, or `Account` could be used to name this concept. The problem is that those can sometimes interfere with the business domain. For example, if we named this concept `Team`, and we're building a Fantasy Football app, we'd want the `Team` model to represent a football team and not a *team* of users. To avoid such confusion, I believe `Organization` is the most appropriate term for this concept[2].

This is not a rule by any means. Depending on your context, you may wish to use another term. It's just worth thinking about whether that term could ever cause confusion in your app in the future.

Hypothetically, if we took this `User` centric approach initially and somewhere down the line we wanted to implement *organizations* in the app, the entire domain model would need to be reworked. All resources would have to be altered to belong to an `Organization` rather than a `User`. Not to mention the vast changes to application logic, all of which would be written with the assumption that resources belong to a `User`. It becomes a massive refactor and not just a new feature[3].

Even though we won't be building any UI to manage an *organization* in Piazza, building it into the domain model future proofs the app.

But does a two sided marketplace need *organizations*?

YES. What if a local shop starting listing their items and they wanted their employees to be able to post and manage their listings? What if Piazza expanded into more of an eBay style marketplace in the future and businesses started both buying and selling using it? Every single app could potentially grow to require *organizations*. It's prudent to build it into the domain model from day 1, even if it isn't reflected in the UI.

A good rule of thumb is that the `User` should be used solely for authentication and the only entities associated with it should be specific to that particular human; for example, their *preferences*. EVERYTHING else should belong to an `Organization`.

Before starting a feature, it's wise to create and work on a different branch.

```
$ git checkout -b users-and-sign-ups
```

3.1.1. Model classes and database tables

A `User` should be able to belong to many `Organization`s and an `Organization` should be able to have many `User`s. This means there's a *many-to-many* association between these two models which requires a *join* table and corresponding model. This model will be named `Membership`.

Generate[4] the models and database migrations for these entities.

Listing 10. Using the Rails generator to generate the models and migrations

```
$ bin/rails g model User name:string email:string:uniq
password_digest:string
$ bin/rails g model Organization
$ bin/rails g model Membership user:references
organization:references
```

As Listing 10 demonstrates, the `User` model has 3 `string` fields for the `name`, `email` and `password_digest`. Storing the password as plain text is a security flaw in case anyone is able to gain unauthorized access to the database, so it's stored in the form of a secure *digest* generated using a library called `bcrypt`. We've also created a unique index for the `email` field so duplicate users cannot be created.

The `Organization` model doesn't have any fields because it won't be used in the UI or contain any business logic in the scope of this book. It's there for future proofing and fields can be added when the time comes!

Lastly, the `Membership` model has foreign key references to the `User` and `Organization` and its singular purpose is to link these two models. In the future, if needs dictate, this model can be expanded to contain information such as which *role* a `User` performs within an `Organization`. That's also out of scope for this book.

Run the database migrations to create the tables.

```
$ bin/rails db:migrate
```

The generators will have created some *fixture* files containing test data that is loaded into the database before every test run. The autogenerated data will cause errors so clear out these files for now. We'll add test data as and when we need it. Clear the contents of these files:

- `test/fixtures/users.yml`
- `test/fixtures/organizations.yml`
- `test/fixtures/memberships.yml`

3.1.2. Validations and relationships

Now that the models are in place, let's add some validations and the relationships to mirror the database foreign keys.

The user must have a `name` and a valid unique `email` so these attributes need to be validated. For `name`, a simple presence check will suffice but the email will be validated against a built-in regex.

This is pretty clear cut logic so let's start with a few tests as shown in Listing 11.

Listing 11. Model tests for name and email validation (**`test/models/user_test.rb`**)

```
class UserTest < ActiveSupport::TestCase
  test "requires a name" do
    @user = User.new(name: "", email: "johndoe@example.com")
    assert_not @user.valid?
    @user.name = "John"
    assert @user.valid?
  end

  test "requires a valid email" do
    @user = User.new(name: "John", email: "")
    assert_not @user.valid?

    @user.email = "invalid"
    assert_not @user.valid?

    @user.email = "johndoe@example.com"
    assert @user.valid?
  end

  test "requires a unique email" do
    @existing_user = User.create(
      name: "John", email: "jd@example.com"
    )
    assert @existing_user.persisted?

    @user = User.new(name: "Jon", email: "jd@example.com")
    assert_not @user.valid?
  end
end
```

Running these tests will fail as we haven't written any application code as yet.

Listing 12. **RED**

```
$ bin/rails test
```

Write the application code to make the tests pass as shown in Listing 13.

Listing 13. Validating the name and email for a user (`app/models/user.rb`*)*

```
class User < ApplicationRecord
```

```
  validates :name, presence: true
  validates :email,
    format: { with: URI::MailTo::EMAIL_REGEXP },
    uniqueness: { case_sensitive: false }
end
```

The test suite will now be green!

Listing 14. **GREEN**

```
$ bin/rails test
```

Now let's set up the relationships between the models to mirror the foreign keys in the database. Listing 15, Listing 16 and Listing 17 show the additions for each of the models.

Listing 15. A `User` *has many* `Organizations` *(*`app/models/user.rb`*)*

```
class User < ApplicationRecord
  # ...

  has_many :memberships, dependent: :destroy
  has_many :organizations, through: :memberships
end
```

The `dependent: :destroy` option for the `memberships` relation in Listing 15 is specified so the corresponding `Membership` is deleted when a `User` is deleted. We'll do the same for `Organization` in Listing 16.

Listing 16. An `Organization` *has many* `Users` *(*`app/models/organization.rb`*)*

```
class Organization < ApplicationRecord
  has_many :memberships, dependent: :destroy
  has_many :members, through: :memberships, source: :user
end
```

In Listing 16, the relation to `User` is defined as `has_many :members` because `@organization.members` reads better than `@organization.users`. The `source: :user` option tells ActiveRecord to use the `user_id` foreign key for this relation.

Listing 17. A `Membership` belongs to a `User` and an `Organization` (**`app/models/membership.rb`**)

```
class Membership < ApplicationRecord
  belongs_to :user
  belongs_to :organization
end
```

3.1.3. Strip extraneous spaces

A common mistake people make when filling web forms is to accidentally include a space at the start or end of a field. This isn't always an issue, but in an email address, an unwanted space can cause all kinds of problems.

We can use the `normalizes` method provided by Active Record to strip extraneous spaces. Let's start with a test once again.

Listing 18. Model tests for removal of extraneous spaces **RED** (**`test/models/user_test.rb`**)

```
class UserTest < ActiveSupport::TestCase
  # ...

  test "name and email is stripped of spaces before saving" do
    @user = User.create(
      name: " John ",
      email: " johndoe@example.com ",
    )

    assert_equal "John", @user.name
    assert_equal "johndoe@example.com", @user.email
  end
```

```
end
```

Listing 19. Normalizing the name and email **GREEN** *(`app/models/user.rb`)*

```
class User < ApplicationRecord
  # ...

  normalizes :name,   with: -> (name) { name.strip }
  normalizes :email,  with: -> (email) { email.strip.downcase }
end
```

3.1.4. Secure passwords

As touched on earlier, the password needs to be stored securely in the database in the form of a *digest*. Rails provides a helper called `has_secure_password` to do exactly this. Before it can be used, the `bcrypt` gem used to compute the digest needs to be added to the project.

Open up your Gemfile. It should be there as a comment.

Listing 20. Uncomment `bcrypt` in the Gemfile (`Gemfile`)

```
# ...

# Use Active Model has_secure_password
gem "bcrypt", "~> 3.1.7"

# ...
```

Install `bcrypt`.

```
$ bundle install
```

Along with securing the password, we'll also add a validation to check its length and presence. We'll set a minimum of 8 characters for security reasons.

Once again, let's start with tests as shown in Listing 21.

Listing 21. Testing password validation **RED** (**`test/models/user_test.rb`**)

```
class UserTest < ActiveSupport::TestCase
  # ...

  test "password length must be between 8 and ActiveModel's maximum"
do
    @user = User.new(
      name: "Jane",
      email: "janedoe@example.com",
      password: ""
    )
    assert_not @user.valid?

    @user.password = "password"
    assert @user.valid?

    max_length =
      ActiveModel::SecurePassword::MAX_PASSWORD_LENGTH_ALLOWED
    @user.password = "a" * (max_length + 1)
    assert_not @user.valid?
  end
end
```

Next, let's go back to the `User` model and add the password validation and `has_secure_password` directive to ensure it's stored securely. The maximum length validation is added automatically by Rails so that doesn't need to be defined.

Listing 22. Securing and validating the password (**`app/models/user.rb`**)

```
class User < ApplicationRecord
  # ...

  before_validation :strip_extraneous_spaces

  has_secure_password
  validates :password,
    presence: true,
    length: { minimum: 8 }

  private

  # ...
end
```

`has_secure_password` packs in a lot with a single line of code. It expects a `password_digest` column in the database (another name can be specified, more on this later) and transparently hashes a password when it's passed in while creating or updating a `User`!

Now let's run the test suite again. It's still **RED**!

Listing 23. The test suite shows 2 failures. **RED**

```
$ bin/rails test
```

The tests checking for a valid name and email address are no longer passing. This is because we've added a new validation for a password, which isn't being passed in when creating a `User` in those tests. That's straightforward enough to fix.

Listing 24. Fix the test failures caused by password validation **GREEN** *(**`test/app/user_test.rb`**)*

```
class UserTest < ActiveSupport::TestCase
  test "requires a name" do
    @user = User.new(
      name: "",
      email: "johndoe@example.com",
      password: "password"
    )
    assert_not @user.valid?

    @user.name = "John"
    assert @user.valid?
  end

  test "requires a valid email" do
    @user = User.new(
      name: "John",
      email: "",
      password: "password"
    )
    assert_not @user.valid?

    @user.email = "invalid"
    assert_not @user.valid?

    @user.email = "johndoe@example.com"
    assert @user.valid?
  end

  test "requires a unique email" do
    @existing_user = User.create(
      name: "John",
      email: "jd@example.com",
      password: "password"
    )
    assert @existing_user.persisted?

    @user = User.new(
      name: "Jon",
      email: "jd@example.com",
      password: "password"
    )

    assert_not @user.valid?
  end

  # ...
end
```

Now's also a good time to commit your code.

```
$ git add .
$ git commit -m "Set up user, organization, and membership models"
```

3.2. Rails and I18n

Before building the sign up form, let's take a slight detour and discuss *internationalization* (`I18n`) and *localization* (`L18n`). Piazza will be internationalized from the get-go. This means that all user facing text will be abstracted out of our application and into *locale dictionaries* so the application can be served in multiple languages.

We'll be building in English in this book, but if Piazza ever needed to be translated into another language, all that would need doing is to get the text translated, drop it into the app and make some minor configuration changes!

Internationalizing from day 1 has other advantages as well. It keeps all user facing text out of the controllers and views making it easier to see the wood for the trees. While writing tests, we can assert for the *key* in the *locale dictionary* rather than the text itself meaning the tests are robust when user facing text changes. `I18n` is a minor trade-off that can pay dividends in the future.

`I18n` in Rails is provided by the Ruby I18n gem. All Rails apps have a basic `I18n` setup out of the box. The user facing text is placed in YAML files in the `config/locales/` folder with the language's ISO code as the root key.

Listing 25. An example of a locale dictionary

```
en:
  hello: Hello World!
  goodbye: Goodbye.
```

These strings can be referenced in code using the `translate` method, or the shorthand `t`.

```
I18n.translate "hello"
# => Hello World!
I18n.t "goodbye"
# => Goodbye.
```

Conceptually, that's all there is to it! `I18n` uses `:en` as the default locale so that doesn't need configuration until more languages are added[5].

3.2.1. Localized strings in views and controllers

The key for a localized string can be inferred from context when `t` is called in a controller or view. For example, to show a *success* flash message, we'd do something like Listing 26.

Listing 26. Referencing a localized string using an inferred key

```
class UsersController < ApplicationController
  def create
    # ...

    redirect_to home_path, notice: t(".success")
  end
end
```

You'll notice the `I18n` module doesn't need to be specified when calling `t` in a controller or view.

You'll also notice the key has a `.` prefix. This tells Rails to infer the full key from context and it will expand to `"users.create.success"`. By convention, the name of the controller and action separated by a `.` is prepended to the supplied key. The same logic applies in view files as well. This way, key names remain succinct!

3.2.2. Localization in Active Record

Active Record attributes are often used in the UI in Rails. For example, a typical Rails form looks something like:

```
<%= form_with(model: @user) do |form| %>
  <%= form.label :name %>
  <%= form.text_field :name %>

  <%= form.label :email %>
  <%= form.email_field :email %>
<% end %>
```

`name` and `email` are attributes on the `User` model. One way to localize these terms is to explicitly pass in a value:

```
<%= form.label :name, value: t(".name") %>
```

But this approach will lead to duplication in the locale dictionaries and be tedious to manage. There's a better way. Active Record has built in localization features for model attributes and even the model name itself. Translations can be stored under the `activerecord` key and will be picked up by translation methods.

Let's say we wanted to surface the `User` model as `"Person"` in the UI and localize all its attributes. The locale dictionary would look like:

```
en:
  activerecord:
    models:
      user: Person
    attributes:
      user:
        name: Name
        email: Email
        password: Password
```

The below methods can be used to access these translations:

```
User.model_name.human
# => Person

User.human_attribute_name(:name)
# => Name
```

That's how Active Record hooks directly into localization and no extra code is needed to localize the models and their attributes. The form helpers conveniently use these translation methods behind the scenes!

This technique is also handy when the user facing value of a model or attribute is different from what it's called in the database.

3.2.3. Folder structure for locale dictionary

As mentioned earlier, the locale dictionaries go in the `config/locales/` folder. Theoretically, we could put all the strings in one massive file, but that'd make it impossible to maintain. Rails does not enforce any structure in this case, so it's prudent to follow some basic rules for organization.

Taking inspiration from the folder structure under `app/`, we'll create two folders called `views` and `models` under `config/locales/`. In these folders, we'll create subfolders for each model or controller, and within those we'll create `.yml` files for each language. As an example, for a `User` model and a `UsersController`, the folder structure would look like Listing 27.

Listing 27. A demo of the folder structure for locale dictionaries

```
locales/
|- models/
|-- user/
|      |-- en.yml
|      |-- de.yml
|- views/
|-- users/
|      |-- en.yml
|     |-- de.yml
```

Don’t worry if this is a bit confusing right now. It’ll become clearer as we go along.

3.2.4. Localizing the page title

Let’s put some of the above ideas straight into practice by extracting the page title into a locale dictionary. Firstly, Rails has a default placeholder locale dictionary file, but we don’t want that so get rid of it.

```
$ rm config/locales/en.yml
```

Each page will optionally supply its own title to help with SEO and accessibility but we’ll include a default as well. To facilitate this, we’ll use the `content_for` helper.

Additionally we’ll define a helper of our own to format the title. This appears in Listing 28.

Listing 28. A helper to assemble the page title (`app/helpers/application_helper.rb`)

```
module ApplicationHelper
  def title
    return t("piazza") unless content_for?(:title)

    "#{content_for(:title)} | #{t("piazza")}"
  end
end
```

Let’s write a test for this helper as well.

Listing 29. Create the test file

```
$ touch test/helpers/application_helper_test.rb
```

Listing 30. Testing the helper for the page title **RED** (**`test/helpers/application_helper_test.rb`**)

```ruby
require 'test_helper'

class ApplicationHelperTest < ActionView::TestCase
  test "formats page specific title" do
    content_for(:title) { "Page Title" }

    assert_equal "Page Title | #{I18n.t('piazza')}", title
  end

  test "returns app name when page title is missing" do
    assert_equal I18n.t('piazza'), title
  end
end
```

The test will fail as we haven't defined the localized string as yet.

In the previous section, we discussed a folder structure for locale dictionaries based around *models* and *views*. The default page title doesn't belong such a context though, it's global across the app. This is why the translation key is not prefixed with a `.`.

This string belongs in a file for globals. Create this file by running the commands in Listing 31 and then fill it in with Listing 32.

Listing 31. Create a locale dictionary for globals

```
$ mkdir config/locales/globals
$ touch config/locales/globals/en.yml
```

Listing 32. Add the translation for Piazza (**`config/locales/globals/en.yml`**)

```yaml
en:
  piazza: Piazza
```

The tests should now pass.

Listing 33. **GREEN**

```
$ bin/rails test
```

Restart the Rails server so it picks up the new locale dictionary. Amend the application layout to use this helper.

Listing 34. Using the new `title` *helper* (**`app/views/layouts/application.html.erb`**)

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%# ... %>
  </head>

  <%# ... %>
</html>
```

The result can't be seen just yet as we haven't created any views or controllers. That's up next!

3.3. The sign up form

We're now ready to write some UI code so we can actually see some of the fruits of our work. As mentioned in Chapter 1 we'll use Bulma for our CSS. Bulma is an intuitive class-based framework so we won't focus on it in this book.

The class names are descriptive but should you need further clarification, the Bulma documentation is fantastic!

3.3.1. Setting up the routes, controllers, and views

The first thing we need is a `UsersController` along with actions to show the sign up form and handle the submission. Piazza's homepage will be a *feed* of product ads. We'll get into the specifics later but let's also create a `FeedController`. The user will be redirected to the *feed* after signing up. Run the below commands to create the controllers, actions, and views:

```
$ bin/rails g controller Users new create
$ bin/rails g controller Feed show
```

The generator will create a `app/views/users/create.html.erb` file that isn't needed.

```
$ rm app/views/users/create.html.erb
```

Next, the routes to these controller actions need to be configured. The root path will be the `show` action in `FeedController`. For the sign up form, it's good to have a user friendly path, so we'll serve it from `/sign_up` rather than the conventional `/users/new`. Open `routes.rb` and replace its contents with Listing 35.

Listing 35. The routes for the sign up form (**`config/routes.rb`**)

```ruby
Rails.application.routes.draw do
  root "feed#show"

  get   "sign_up",  to: "users#new"
  post  "sign_up",  to: "users#create"
end
```

If you visit http://localhost:3000/ and http://localhost:3000/sign_up in your web browser, you should see a bit of text with the location of the view file. We're up and running!

Building out the form

Figure 19 shows a wireframe of the sign up form.

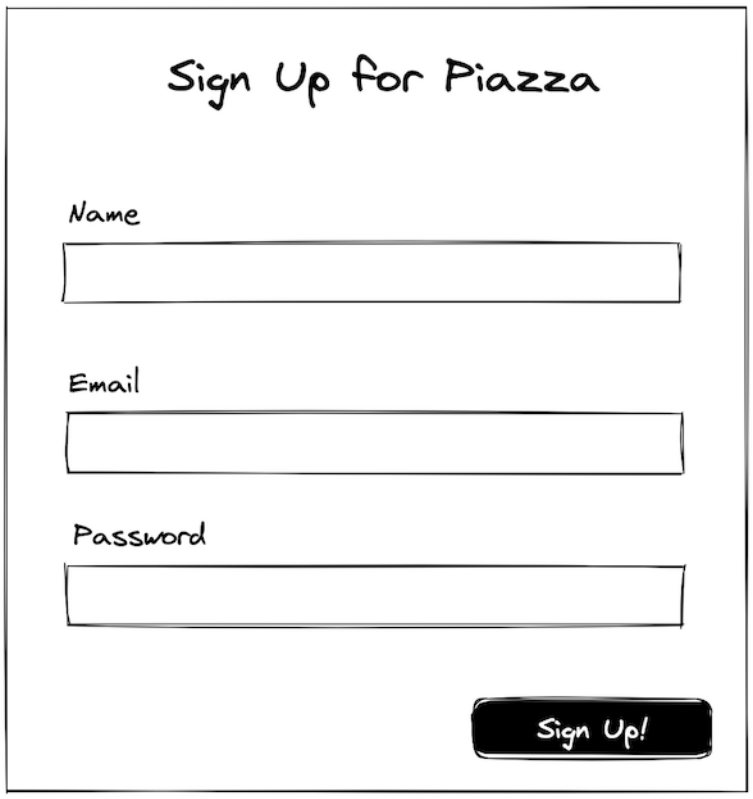


Figure 19. A wireframe of the sign up form

Based on the wireframe, we can write the markup for the form as shown in Listing 36.

Listing 36. The markup for the sign up form (`app/views/users/new.html.erb`)

```
<% content_for :title, t(".title") %>

<layout-columns class="columns is-centered">
  <layout-column class="column box is-5 mt-6 p-5 m-4">
    <h1 class="title has-text-centered">
      <%= t(".title") %>
    </h1>
    <%= form_with(
          model: @user,
          url: sign_up_path,
          class: "is-flex is-flex-direction-column"
        ) do |form| %>
      <div class="block">
        <%= form.label :name, class: "label" %>
        <%= form.text_field :name, class: "input" %>
      </div>

      <div class="block">
        <%= form.label :email, class: "label" %>
        <%= form.email_field :email, class: "input" %>
      </div>

      <div class="block">
        <%= form.label :password, class: "label" %>
        <%= form.password_field :password, class: "input" %>
      </div>

      <%= form.submit t(".sign_up"),
            class: "button is-primary is-large
              is-align-self-flex-end mb-3 mt-3" %>
    <% end %>
  </layout-column>
</layout-columns>
```

This code won't work as the `@user` instance variable used to build the form hasn't been defined in the controller. We're also using a couple of localized strings which don't exist in the locale dictionary.

A quick sidebar on semantic HTML tags

You may have noticed the `layout-columns` and `layout-column` tags in Listing 36. These are semantic HTML tags. HTML supports any custom tags as long as they contain a `-`. They don't need to be defined or registered anywhere and all their layout behavior comes from the `class`es attached to them. I consider it good practice to use semantic HTML tags wherever possible so the markup doesn't become a unwieldy bowl of `div` soup[6].

This kind of custom tags are primarily used for JavaScript custom elements, but there's no compulsion for that. Custom tags can be used to make HTML read better and we'll be doing that throughout this book.

Moving on to the `UsersController`, implement the two actions as shown in Listing 37.

Listing 37. The controller actions for sign ups (**`app/controllers/users_controller.rb`**)

```ruby
class UsersController < ApplicationController
  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)

    if @user.save
      @organization = Organization.create(members: [@user])
      # TODO: Log in user...

      redirect_to root_path,
        status: :see_other,
        flash: { success: t(".welcome", name: @user.name) }
    else
      render :new, status: :unprocessable_entity
    end
  end

  private
    def user_params
      params.require(:user).permit(:name, :email, :password)
    end
end
```

The new action initializes a new instance of User to build the sign up form. The create action creates a new User from sanitized parameters along with a companion Organization. In the next chapter, we'll build an authentication and login system, so a TODO comment marks where the login logic will go.

HTTP statuses for redirects and errors

You'll notice in Listing 37, the response status for both redirect_to and render in the create action is explicitly defined. These are required by Turbo. All form submissions must redirect with a 303 See Other if nothing goes wrong, or render with a 4xx or 5xx if something does.

For redirects, Rails uses 302 Found by default. The specification for this response[7] states that the redirected request should use the same HTTP method as the original request. Due to legacy reasons, the fetch (used by Turbo under the hood) implementation in most browsers will respond to a 302 Found response from a POST request by issuing a GET request to the redirected path. For all other HTTP methods, it will use the same method as the original request when redirecting. Responding with a 303 See Other status ensures that the method used for the redirect is always GET.

This anomaly with redirect codes isn't a massive problem in Rails because Rails forms use the only two browser native methods: GET and POST. Forms with different methods are rendered to use POST with the method inserted as a hidden field which is then parsed by Rails before the request hits a controller. This could potentially change as Rails integrates more tightly with Turbo and it's best to follow HTTP conventions anyway. In this book we'll specify the redirect status as 303 See Other when the redirected request is expected to be a GET.

More information is available in the Turbo docs: https://turbo.hotwired.dev/handbook/drive#redirecting-after-a-form-submission

Next, let's put in the localized strings used in the sign up form. As discussed in Section 3.2.3, create the locale dictionary files for the `User` model and `UsersController`:

```
$ mkdir config/locales/models
$ mkdir config/locales/views
$ mkdir config/locales/models/user
$ mkdir config/locales/views/users

$ touch config/locales/models/user/en.yml
$ touch config/locales/views/users/en.yml
```

Then insert the user facing strings in those files as shown in Listing 38 and Listing 39.

Listing 38. The locale dictionary for the sign up form (`config/locales/views/users/en.yml`)

```
en:
  users:
    new:
      title: Sign up for Piazza
      sign_up: Sign up!
    create:
      welcome: "Welcome to Piazza, %{name}"
```

Listing 39. An empty model locale dictionary for future use (`config/locales/models/user/en.yml`)

```
en:
```

We've put in the two strings that we're using in the sign up form, as well as the flash message shown after a successful sign up. The latter case also demonstrates how to interpolate values in localized strings.

Restart your Rails server so it picks up the newly created locale dictionaries. Then visit http://localhost:3000/sign_up and you'll see the shiny new sign up form as shown in Figure 20.

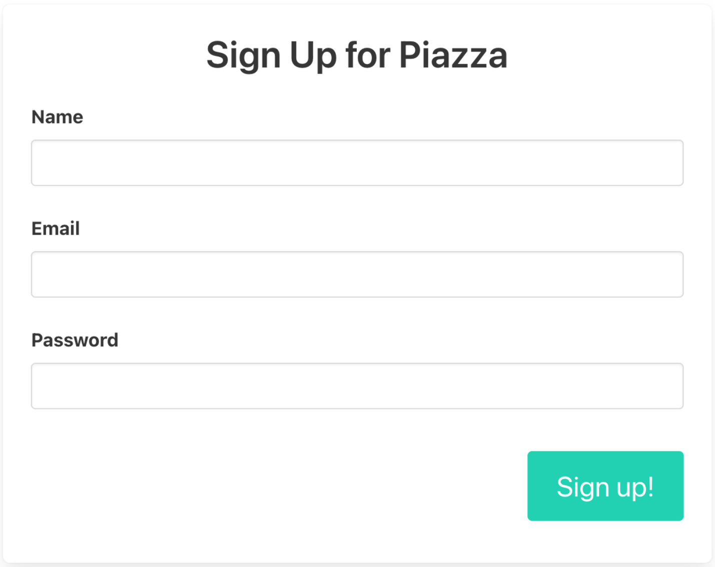


Figure 20. The Sign Up form

Go ahead and fill it with some valid data and you'll be redirected to the root path. But hang on, where the welcome message? We've stored it in the `flash` but it still needs to be rendered in the UI.

Rendering flash messages

Rails' `flash` mechanism is used to render alerts throughout the app, so it belongs in the main `application.html.erb` layout file.

Listing 40. Render a partial with flashes in the application layout (`app/views/layouts/application.html.erb`)

```
<!DOCTYPE html>
<html>
  <%# ... %>

  <body>
    <%= render "application/flashes" %>
    <main>
      <%= yield %>
    </main>
  </body>
</html>
```

Next, create and fill in the flashes partial.

```
$ mkdir app/views/application
$ touch app/views/application/_flashes.html.erb
```

Listing 41. The flashes partial (**`app/views/application/_flashes.html.erb`**)

```
<%# locals: () -%>

<div class="container is-fluid" data-turbo-temporary>
  <% flash.each do |level, message| %>
    <%= tag.div \
          class: "notification is-#{level} is-light
            has-text-centered mt-4" do %>
      <%= message %>
    <% end %>
  <% end %>
</div>
```

Strict locals in partials

Rails 7.1 introduced a mechanism to explicitly define the local variables used by a partial.

A *magic comment* at the top of a partial's file defines which variables it accepts and optional default values for them.

Local variables can be explicitly defined as:

```
<%# locals: (name:, position:, last_signed_in:) -%>
```

A default value can also be specified:

```
<%# locals: (name:, position:, last_signed_in: nil) -%>
```

Or we can disallow all local variables:

```
<%# locals: () -%>
```

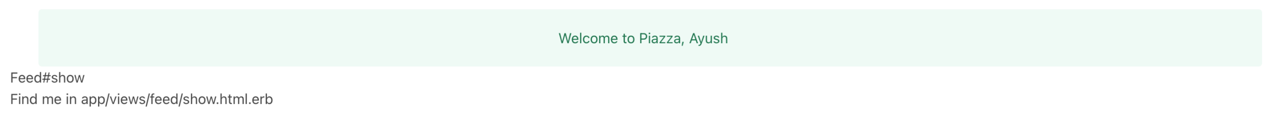
You can see this in action in Listing 41.

This mechanism is great for preventing bugs as well as documentation. Passing in variables not explicitly defined will raise an error, as will omitting required variables.

Using them is optional, but in this book we'll use them at every available opportunity as it makes the app more robust.

`data-turbo-temporary` is specified on the container element to prevent Turbo from caching the alert during navigation. Without this attribute, users will see the alert for a fraction of a second before it disappears if they navigate back to a page with a flash after having navigated away due to Turbo's local cache.

Now if you go back to your sign up form and submit it with some valid data, you'll see an alert as depicted in Figure 21.


Welcome to Piazza, Ayush

Feed#show
Find me in app/views/feed/show.html.erb

Figure 21. A successful sign up

The form works great with valid data, but what about invalid data?

3.3.2. Rendering form input errors

Every web app should be designed to cater for unexpected and invalid user input. Signing up for Piazza with a duplicate email or a password shorter than 8 characters should result in an error. But right now, nothing happens. Well, almost nothing.

If you submit the form with a password that's too short and then view the HTML source, you'll see Listing 42 when inspecting the password field.

Listing 42. The HTML after submitting an invalid form

```
<div class="block">
  <div class="field_with_errors">
    <label class="label" for="user_password">Password</label>
  </div>
  <div class="field_with_errors">
    <input class="input" type="password" name="user[password]"
id="user_password">
  </div>
```

```
</div>
```

Rails has wrapped the fields that contain errors in a `<div>` with the class `field_with_errors`. Rendering errors in this way doesn't work for us because Bulma has no concept of `field_with_errors`. It uses a class called `is-danger` to denote errors, so we need to modify the default behavior for form errors.

`ActionView::Base` has a `field_error_proc` field that can be set to customize the output of form fields that contain errors. The best place to do this is in an *initializer* that goes in `config/initializers/`. Every file in this folder is executed when Rails boots. Create a new file in that folder:

```
$ touch config/initializers/form_errors.rb
```

The `field_error_proc` is executed within the context of the `ActionView::Base` instance, meaning we can render a partial inside it.

Listing 43. Customize the output of form fields with errors (**`config/initializers/form_errors.rb`**)

```
ActionView::Base.field_error_proc = proc do |html_tag, instance|
  render "application/form_errors",
    html_tag: html_tag, instance: instance
end
```

Create the partial and fill it in with Listing 44.

```
$ touch app/views/application/_form_errors.html.erb
```

Listing 44. The partial for form errors (**`app/views/application/_form_errors.html.erb`**)

```
<%# locals: (html_tag:, instance:) -%>

```

```
<% unless html_tag =~ /^<label/ %>
  <% field_html = Nokogiri::HTML::DocumentFragment.parse(html_tag) %>
  <% field_html.children.add_class("is-danger") %>

  <%== field_html %>
  <p class='help is-danger'>
    <%= instance.error_message.to_sentence %>
  </p>
<% else %>
  <%= html_tag %>
<% end %>
```

Let's go through this process step-by-step.

The `field_error_proc` will be called by the *form builder* (by default, an instance of `ActionView::Helpers::FormBuilder`) when rendering a field containing an error. Two arguments are passed to it which are also passed through to the partial in Listing 44.

The first is the complete HTML tag for the field that has an error. For example, if the password field had an error, the below string would be passed as the `html_tag`:

```
"<input class='input' type='password' name='user[password]'
id='user_password'>"
```

The second argument is an instance of the *field* from the `FormBuilder` for the form, which is a subclass of `ActionView::Helpers::Tags::Base`. For a password field, it will be an instance of `ActionView::Helpers::Tags::PasswordField`.

Knowing the information that's passed to the partial, we can move on to the implementation. This proc is called for the problematic attribute's `<label>` as well as `<input>`. The error needs to be shown only on the `<input>`, so using a regex test, the `<label>` is ignored.

For all other tags, the `is-danger` class needs to be added for Bulma to render it in an error state. Using a library called *Nokogiri*[8] which is included in Rails, we parse and manipulate the HTML.

```
<% field_html = Nokogiri::HTML::DocumentFragment.parse(html_tag) %>
<% field_html.children.add_class("is-danger") %>
<%== field_html %>
```

Using `<%==` instead of `<%=` prevents the HTML from being *escaped* so it executes and doesn't render on screen as code.

We also need to render the error message. This is available on the `instance` passed to the partial, so we wrap it in a `<p>` tag and render it out.

```
<p class='help is-danger'>
  <%= instance.error_message.to_sentence %>
</p>
```

Restart your Rails server so this initializer is executed. Then submit the sign up form with invalid data again and you'll see errors being rendered on screen.

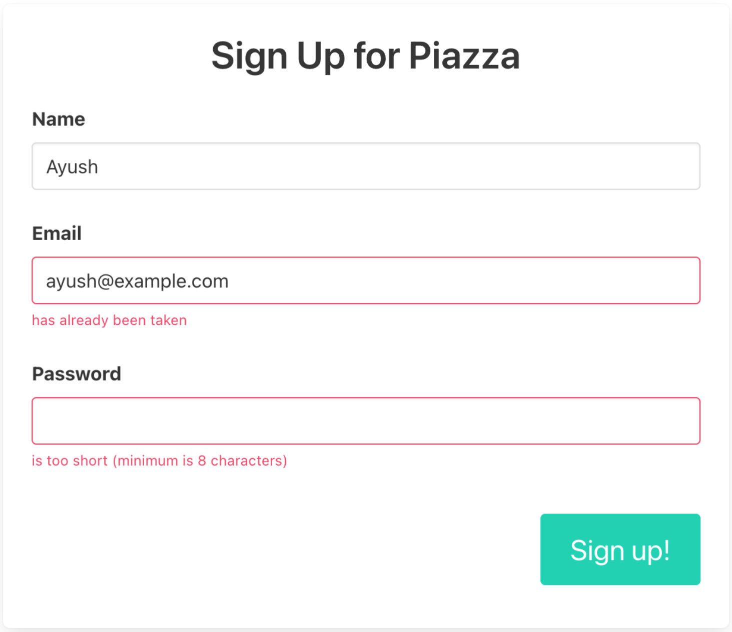


Figure 22. Error messages in forms

Customizing Active Record error messages

The error messages seen in the form come from Active Record. They're generated when model validations are not satisfied. As such, they can be customized or localized in the same way as Active Record model or attribute names.

The easiest way to determine the key for any given string is to use a gem called `i18n-debug`. It logs the locale key for every string rendered in a view to the console.

Add this gem to the `development` group in your `Gemfile` as it's not needed outside the local environment.

*Listing 45. Add `i18n-debug` to the Gemfile (**`Gemfile`**)*

```
# ...

group :development do
  # ...

  gem "i18n-debug"
end
```

Then run:

```
$ bundle install
```

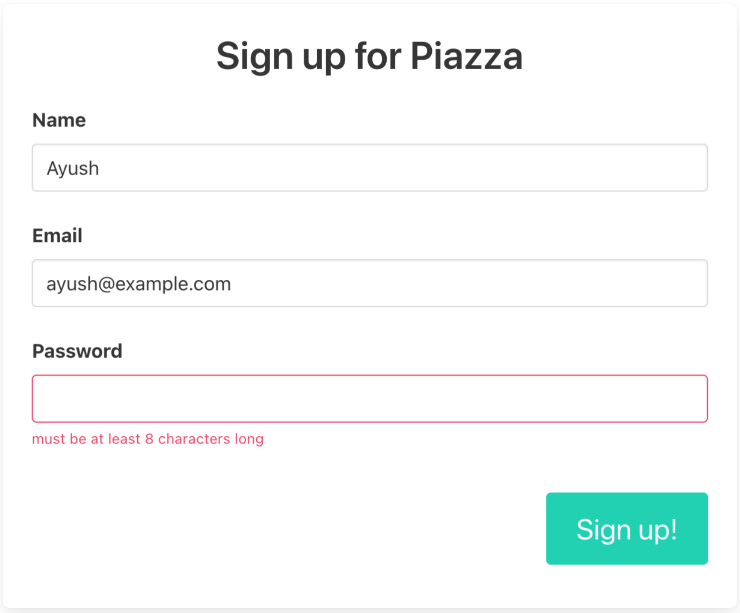
Restart your Rails server and once again submit the form with a password that's too short. In the server logs in your Terminal, you'll see the locale keys of every string rendered in the view.

Looking closely at these keys, you'll see: `en.activerecord.errors.models.user.attributes.password.too_short`. This is the key to localize the message for a short password error. Go ahead and do that as shown in Listing 46.

Listing 46. A localized error message (`config/locales/models/user/en.yml`)

```yaml
en:
  activerecord:
    errors:
      models:
        user:
          attributes:
            password:
              too_short: must be at least 8 characters long
```

Once again, submit the form with a password that's too short, you'll see the new error message.


Sign up for Piazza

Name: Ayush

Email: ayush@example.com

Password:

must be at least 8 characters long

Figure 23. The new custom error message

3.3.3. Controller testing

With the sign up flow is in place, it's a good idea to wrap it in some controller tests. These tests are focused on a single controller and make HTTP requests to a path and assert the responses.

But shouldn't we write the tests first?

Perhaps. If we were staunchly following Test Driven Development (TDD). This is a polarizing topic and a lot of people have strong opinions on when tests should be written. Personally, I struggle to write tests first when the picture of the application code isn't completely clear in my head. For example, when I'm hacking away on a new problem.

In this book we'll write tests first on some occasions when the application behavior is clear and other times we'll write the tests after the application code.

When you're working on your own projects, I recommend finding a method that works for you and following that. Writing tests is essential. When you write them is irrelevant.

For the `UsersController` we'll test that the sign up form can be submitted successfully and shows errors for invalid data.

The Rails generator will have generated a couple of placeholder tests. Clear out their contents and replace them with Listing 47 and Listing 48.

Listing 47. Controller tests for the sign up flow **GREEN** (**`test/controller/users_controller_test.rb`**)

```ruby
require "test_helper"

class UsersControllerTest < ActionDispatch::IntegrationTest
  test "redirects to feed after successful sign up" do
    get sign_up_path
    assert_response :ok

    assert_difference [ "User.count", "Organization.count" ], 1 do
      post sign_up_path, params: {
        user: {
          name: "John",
          email: "johndoe@example.com",
          password: "password"
        }
      }
    end

    assert_redirected_to root_path
    follow_redirect!
    assert_select ".notification.is-success",
      text: I18n.t("users.create.welcome", name: "John")
  end

  test "renders errors if input data is invalid" do
    get sign_up_path
    assert_response :ok

    assert_no_difference [ "User.count", "Organization.count" ] do
      post sign_up_path, params: {
        user: {
          name: "John",
          email: "johndoe@example.com",
          password: "pass"
        }
      }
    end

    assert_response :unprocessable_entity
    assert_select "p.is-danger",
      text:
        I18n.t
("activerecord.errors.models.user.attributes.password.too_short")
  end
end
```

Listing 48. Delete the placeholder tests for the `FeedController` (**`test/controller/feed_controller_test.rb`**)

```
require "test_helper"

class FeedControllerTest < ActionDispatch::IntegrationTest
```

```
end
```

The two test cases in Listing 47 simulate the HTTP requests made when a user submits the sign up form first with valid data and then with a password that doesn't meet the requirements.

You can also see how the text on the page is asserted using the locale key. If the user facing text changes, the tests won't break as they're effectively testing against a variable. Neat!

The full list of test actions available in controller tests are available in the Rails docs: https://api.rubyonrails.org/classes/ActionDispatch/Integration/RequestHelpers.html.

Assertion methods tailored for controller testing live in their own repository called `rails-dom-testing`. The docs are available here: https://www.rubydoc.info/gems/rails-dom-testing/.

3.4. Conclusion

That brings us to the end of this chapter! We've covered a lot of ground so let's recap what we've built so far.

1. Created the `User`, `Organization` and `Membership` models that form the backbone of Piazza.
2. Added validations and tests to the above models.
3. Stored passwords securely in the database.
4. Discussed how `I18n` and `L18n` works in Rails and why it's a good idea to *internationalize* single language apps.
5. Built a sign up form and covered it with controller tests.

Commit and push your code. This will also trigger a deploy on Render, so you can go to your app's URL (visible in the Render dashboard) and play around with your sign up form in a production setting!

```
$ git add .
$ git commit -m "Complete sign up form"
$ git checkout main
$ git merge users-and-sign-ups
$ git push
```

3.4.1. Exercises

There are several exercises throughout this book. In most cases, the exercise solutions are tangential to the main text, but sometimes they involve fixing a failing test suite which is required to proceed with the text. I recommend completing all exercises before proceeding.

1. Add a password confirmation field to the sign up form. `has_secure_password` supports both a `password` and `password_confirmation` parameter input and compares the two before saving the record. This makes it trivial to add a password confirmation field. Ensure a password mismatch error is shown to the user if their passwords don't match. Ensure you write a controller test for this as well.

[1] MVP stands for Minimum Viable Product. More information is available here: https://www.agilealliance.org/glossary/mvp/.

[2] This article by Andrew Culver describes why "Organization" is the most appropriate term to use and why it's essential in every web app: https://blog.bullettrain.co/teams-should-be-an-mvp-feature/.

[3] In one of my previous jobs at a billion dollar startup, this refactor took several months and was a massive and challenging project because the app had reached significant size and scale without the concept of organizations. All the resources were tied to individual users and introducing a business offering meant refactoring all this code before even getting started with any new business facing features. To avoid such scenarios, the initial added complexity of an Organization is a worthy trade-off!

[4] Rails has an incredibly convenient "generate" command that we're using to create these models and database migrations, and will be using throughout this book. The syntax can be hard to remember so I recommend using http://rails.help to generate the "generate" commands!

[5] There's a comprehensive Rails guide that covers all things I18n in Rails: https://guides.rubyonrails.org/i18n.html.

[6] For more information about semantic HTML tags, this excellent blog post from Jared White has the complete lowdown: https://www.bridgetownrb.com/showcase/custom-html-elements-everywhere-for-page-layout/.

[7] More information about the 302 response is can be found in the MDN docs: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/302.

[8] Nokogiri is the most popular Ruby library for dealing with HTML. The complete docs are available here: https://nokogiri.org.

Thanks for for downloading the preview.