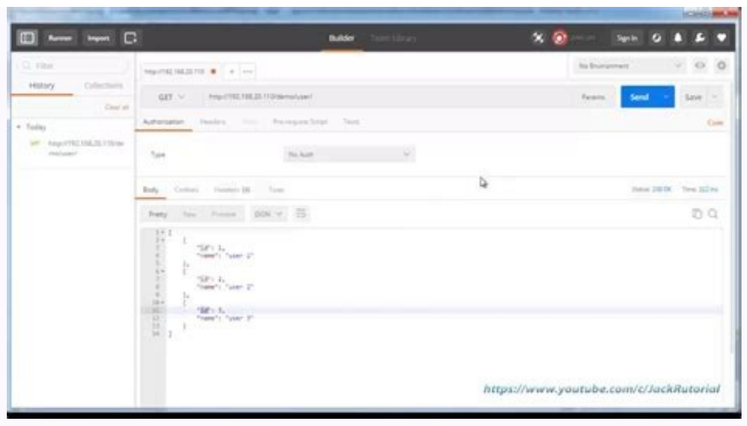
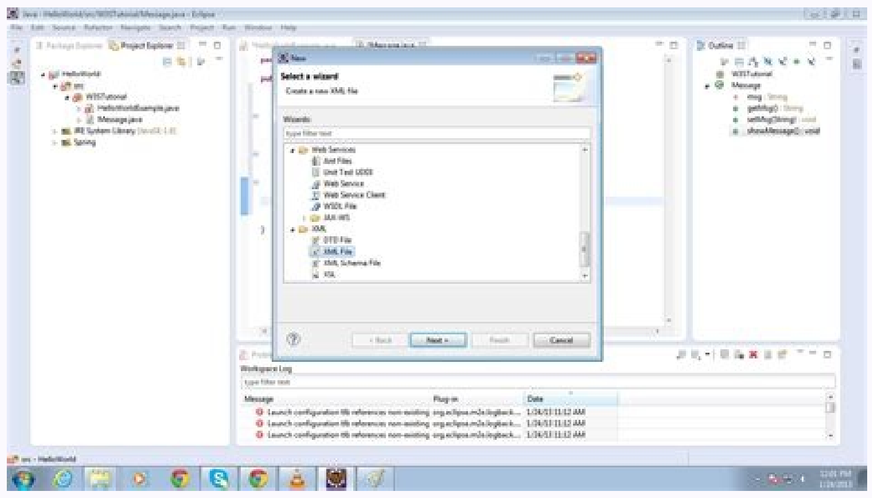
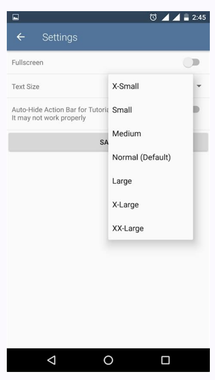
Spring android tutorial

Continue







You already know that building APIs with Spring Boot is incredibly easy. But, your API isn’t complete without a UI, right? Well, building UIs with Ionic is pretty easy too, especially if you know Angular! Ionic is an open source framework designed to help you build mobile applications with web technologies. It started out as a framework based on AngularJS. Ionic 3.0 was recently released, with support for Angular 4, TypeScript 2.2, and lazy loading. When developing an Ionic app, you’ll use Angular and have access to native APIs via Ionic Native and Apache Cordova. This means you can develop slick-looking UIs using the technologies you know and love: HTML, CSS, and TypeScript. This tutorial will show you how to build a Spring Boot API and an Ionic app, and then how to deploy the finished product to your iOS or Android-based mobile device. Together, these technologies provide an excellent developer experience, with hot reloading so you see immediate feedback whenever you change files. Prerequisites: Java 8 and Node.js installed. Create a New Spring Boot Project To begin, create a directory on your hard drive called spring-boot-ionic-example. During this tutorial, you will create server and ionic-beer directories to hold the server and client applications, respectively. Build a Spring Boot API I recently wrote about how to build a Spring Boot API in Bootiful Development with Spring Boot with Angular. Rather than covering that again, you can clone the existing project and copy the server directory into spring-boot-ionic-example. git clone cp -r spring-boot-angular-example/server ~/spring-boot-ionic-example/. This project contains a beers API that allows you to CRUD a list of beer names. It also contains a /good-beers endpoint that filters out less-than-great beers. A BeerCommandLineRunner class creates the default list of beers: @Component class BeerCommandLineRunner implements CommandLineRunner { private final BeerRepository repository; public BeerCommandLineRunner(BeerRepository repository) { this.repository = repository; } @Override public void run(String... strings) throws Exception { // Top beers from Stream.of("Kentucky Brunch Brand Stout", "Good Morning", "Very Hazy", "King Julius", "Budweiser", "Coors Light", "PBR").forEach(name -> repository.save(new Beer(name))); repository.findAll().forEach(System.out::println); } } The BeerRepository interface is decorated with @RepositoryRestResource to expose CRUD endpoints for the Beer entity. @RepositoryRestResource interface BeerRepository extends JpaRepository {} The last piece of the API is the BeerController that exposes /good-beers and specifies cross-origin resource sharing (CORS) settings. @RestController public class BeerController { private BeerRepository repository; public BeerController(BeerRepository repository) { this.repository = repository; } @GetMapping("/good-beers") @CrossOrigin(origins = " ") public Collection goodBeers() { return repository.findAll().stream() .filter(this::isGreat) .collect(Collectors.toList()); } private boolean isGreat(Beer beer) { return !beer.getName().equals("Budweiser") && !beer.getName().equals("Coors Light") && !beer.getName().equals("PBR"); } } You should be able to start the server application by running it in your favorite IDE or from the command line using mvn spring-boot:run. You’ll want to do this from the server directory. If you don’t have Maven installed, you can use the Maven wrapper that’s included in the project (./mvnw spring-boot:run). After the app has started, navigate to . You should see the list of good beers in your browser. Create an Ionic App To create an Ionic app to display data from your API, you’ll first need to install Ionic CLI and Cordova: npm install -g ionic cordova The Ionic CLI is a command-line tool that significantly reduces the time it takes to develop an Ionic app. It’s like a Swiss Army Knife: It brings together a bunch of miscellaneous tools under a single interface. The CLI contains a number of useful commands for Ionic development, such as start, build, generate, serve, and run. After installation completes, cd into spring-boot-ionic-example and create a new application using the following command: You will be prompted to select a starter project and optionally integrate with Cordova to target native iOS and Android. For the purposes of this tutorial, choose the tabs starter project and answer Yes to Cordova. cd ionic-beer ionic serve This will open your default browser on . You can click through the tabbed interface to see the default structure of the app. Open the ionic-beer project in your preferred IDEA to start creating your UI. I recommend using IntelliJ IDEA because it has good TypeScript support and can auto-import classes just like it does for Java project. Create a Good Beers UI Run ionic generate page beer to create a component and a template to display the list of good beers. This creates a number of files in src/pages/beer: beer.html beer.module.ts beer.scss beer.ts Open src/app/app.module.ts and add BeerPageModule to the imports list in app.module.ts. import { BeerPageModule } from '../pages/beer/beer.module'; @NgModule({ ... imports: [BrowserModule, IonicModule.forRoot(MyApp), BeerPageModule], ... }) Create src/providers/beer-service.ts to hold a BeerService that fetches the beer list from the Spring Boot API. Populate it with constants for the API path and add a getGoodBeers() method. import { Injectable } from '@angular/core'; import { Observable } from 'rxjs'; import { HttpClient } from '@angular/common/http'; @Injectable() export class BeerService { public API = ' '; public BEER_API = this.API + '/beers'; constructor(public http: HttpClient) { } getGoodBeers(): Observable { return this.http.get(this.API + '/good-beers'); } } Replace the HTML in src/pages/beer/beer.html to show the list of beers. Good Beers {{beer.name}} Modify src/pages/beer/beer.module.ts to import BeerService and add it as a provider. You could add it as a provider in each component, but adding it in the module allows all components to use it. import { BeerService } from '../../providers/beer-service'; @NgModule({ ... providers: [BeerService] }) Update src/pages/beer/beer.ts to import BeerService and add it as a dependency in the constructor. Call the getGoodBeers() method in the ionViewDidLoad() lifecycle method. import { Component } from '@angular/core'; import { IonicPage, NavController, NavParams } from 'ionic-angular'; import { BeerService } from '../../providers/beer-service'; @IonicPage() @Component({ selector: 'page-beer', templateUrl: 'beer.html' }) export class BeerPage { private beers: Array; constructor(public navCtrl: NavController, public navParams: NavParams, public beerService: BeerService) { } ionViewDidLoad() { this.beerService.getGoodBeers().subscribe(beers => { this.beers = beers; }); } Modify beer.ts to import GiphyService and set a giphyUrl on each beer. import { Component } from '@angular/core'; import { IonicPage, NavController, NavParams } from 'ionic-angular'; import { BeerService } from '../../providers/beer-service'; import { GiphyService } from '../../providers/giphy-service'; @IonicPage() @Component({ selector: 'page-beer', templateUrl: 'beer.html' }) export class BeerPage { private beers: Array; constructor(public navCtrl: NavController, public navParams: NavParams, public beerService: BeerService, public giphyService: GiphyService) { } ionViewDidLoad() { this.beerService.getGoodBeers().subscribe(beers => { this.beers = beers; for (const beer of this.beers) { this.giphyService.get(beer.name).subscribe(url => { beer.giphyUrl = url }); } }) } } Update beer.html to display the image retrieved: {{beer.name}} Start the Spring Boot app in one terminal window and run ionic serve in another. Open in your browser. Click on the Beer icon, and you’ll likely see an error in your browser. Error: Uncaught (in promise): Error: StaticInjectorError[HttpClient]: StaticInjectorError[HttpClient]: NullInjectorError: No provider for HttpClient! This highlights one of the slick features of Ionic: errors display in your browser, not just the browser’s console. Add HttpClientModule to the list of imports in src/app/app.module.ts to solve this issue. import { HttpClientModule } from '@angular/common/http'; @NgModule({ ... imports: [BrowserModule, HttpClientModule, IonicModule.forRoot(MyApp), BeerPageModule], After making this change, you’ll likely see the following error in your browser’s console. XMLHttpRequest cannot load . No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin ' ' is therefore not allowed access. The response had HTTP status code 401. To fix this, open your Spring Boot application’s BeerController.java class and change its @CrossOrigin annotation to allow and . This enables cross-origin resource sharing (CORS) from both the browser and the mobile client (is used by iOS Simulator). @CrossOrigin(origins = {" "," "}) public Collection goodBeers() { Recompile this class, and DevTools should restart the application. If everything works as expected, you should see a page similar to the one below in your browser. Change the header in beer.html to have a button that opens a modal to add a new beer. Good Beers In this same file, change to have a click handler for opening the modal for the current item. Add ModalController as a dependency in beer.ts and add an openModal() method. import { IonicPage, ModalController, NavController, NavParams } from 'ionic-angular'; export class BeerPage { private beers: Array; constructor(public navCtrl: NavController, public navParams: NavParams, public beerService: BeerService, public giphyService: GiphyService, public modalCtrl: ModalController) { } // ionViewDidLoad() { openModal(beerId) { let modal = this.modalCtrl.create(BeerModalPage, beerId); modal.present(); // refresh data after modal dismissed modal.onDidDismiss(() => this.ionViewDidLoad()) } } This won’t compile because BeerModalPage doesn’t exist. Create beer-modal.ts in the same directory. This page will retrieve the beer from the beerId that’s passed in. It will render the name, allow editing, and show the Giphy image found for the name. import { BeerService } from '../../providers/beer-service'; import { Component, ViewChild } from '@angular/core'; import { GiphyService } from '../../providers/giphy-service'; import { NavParams, ViewController, ToastController, NavController } from 'ionic-angular'; import { NgForm } from '@angular/forms'; @Component({ templateUrl: './beer-modal.html' }) export class BeerModalPage { @ViewChild('name') name; beer: any = {}; error: any; constructor(public beerService: BeerService, public giphyService: GiphyService, public params: NavParams, public viewCtrl: ViewController, public toastCtrl: ToastController, public navCtrl: NavController) { if (this.params.data.id) { this.beerService.get(this.params.get('id')).subscribe((beer: any) => { this.beer = beer; this.beer.href = beer._links.self.href; this.giphyService.get(beer.name).subscribe(url => beer.giphyUrl = url); }); } } dismiss() { this.viewCtrl.dismiss(); } save(form: NgForm) { let update: boolean = form['href']; this.beerService.save(form).subscribe(result => { let toast = this.toastCtrl.create({ message: 'Beer "' + form.name + '" ' + ((update) ? 'updated' : 'added') + '.', duration: 2000 }); toast.present(); this.dismiss(); }, error => this.error = error) } ionViewDidLoad() { setTimeout(() => { this.name.setFocus(); },150); } } Add the import for BeerModalPage to beer.ts, then create src/pages/beer/beer-modal.html as a template for this page. {{beer ? 'Beer Details' : 'Add Beer'}} Cancel {{error}} Save You’ll also need to modify beer-service.ts to have get() and save() methods. get(id: string) { return this.http.get(this.BEER_API + '/' + id); } save(beer: any): Observable { let result: Observable; if (beer['href']) { result = this.http.put(beer.href, beer); } else { result = this.http.post(this.BEER_API, beer) } return result.catch(error => Observable.throw(error)); } At this point, if you try to add or edit a beer name, you’ll likely see an error in your browser’s console. Uncaught (in promise): Error: No component factory found for BeerModalPage. Did you add it to @NgModule.entryComponents? To fix this problem, add BeerModalPage to the declarations and entryComponent lists in beer.module.ts. import { BeerModalPage } from './beer-modal'; @NgModule({ declarations: [BeerPage, BeerModalPage], ... entryComponents: [BeerModalPage] }) Now if you try to edit a beer’s name, you’ll see another CORS in your browser’s console. Add a @CrossOrigin annotation to BeerRepository.java (in your Spring Boot project) that matches the one in BeerController. @RepositoryRestResource @CrossOrigin(origins = {" "," "}) Re-compile and now everything should work as expected. For example, below is a screenshot that shows I added a new beer and what it looks like when editing it. Add Swipe to Delete To add swipe-to-delete functionality on the list of beers, open beer.html and make it so wraps and contains the *ngFor. Add a delete button using . {{beer.name}} Delete Add a remove() method to beer.ts. remove(beer) { this.beerService.remove(beer.id).subscribe(response => { for (let i = 0; i < this.beers.length; i++) { if (this.beers[i] === beer) { this.beers.splice(i, 1); let toast = this.toastCtrl.create({ message: 'Beer "' + beer.name + '" deleted.', duration: 2000, position: 'top' }); toast.present(); } } }); } Add and import ToastController as a dependency in the constructor, so everything compiles. constructor(public navCtrl: NavController, public navParams: NavParams, public beerService: BeerService, public giphyService: GiphyService, public modalCtrl: ModalController, public toastCtrl: ToastController) { } You’ll also need to modify beer-service.ts to have a remove() method. remove(id: string) { return this.http.delete(this.BEER_API + '/' + id); } After making these additions, you should be able to delete beer names. To emulate a left swipe in your browser, click on the item and drag it to the left. PWAs with Ionic Ionic ships with support for creating progressive web apps (PWAs). If you’d like to learn more about what PWAs are, see Navigating the World of Progressive Web Apps with Ionic 2. This blog post is still relevant for Ionic 3. If you run the Lighthouse Chrome extension on this application, you’ll likely get a mediocre score in the 40s. To register a service worker, and improve the app’s score, uncomment the following block in src/index.html. After making this change, the score should improve. In my tests, it increased to 73/100. The remaining issues were: Does not provide fallback content when JavaScript is not available. Does not redirect HTTP traffic to HTTPS Page load is not fast enough on 3G Has a tag with width or initial-scale To fix the first issue, add the following HTML in src/index.html, just after the tag. You must enable JavaScript to view this page. To fix the last issue, remove viewport-fit=cover, from the tag’s content attribute. Deploy to a Mobile Device It’s pretty cool that you’re able to develop mobile apps with Ionic in your browser. However, it’s nice to see the fruits of your labor and see how awesome your app looks on a phone. It really does look and behave like a native app! To see how your application will look on different devices you can run ionic serve --lab. The --lab flag opens a page in your browser that lets you see how your app looks on different devices. iOS To emulate or deploy to an iOS device, you’ll need a Mac and a fresh installation of Xcode. If you’d like to build iOS apps on Windows, Ionic offers an Ionic Package service. Make sure to open Xcode to complete the installation. Then run ionic cordova emulate ios to open your app in Simulator. TIP: The biggest problem I found when running the app in Simulator was that it was difficult to get the keyboard to popup. To workaround this, I used Hardware > Keyboard > Toggle Software Keyboard when I needed to type text in a field. Deploying to your phone will likely fail because it won’t be able to connect to . To fix this, you can deploy your Spring Boot app to a public server, or use your computer’s IP address in beer.service.ts (if you’re on the same wireless network). To deploy to Cloud Foundry, copy this deploy.sh script to your hard drive. It expects to be in a directory above your apps (e.g. spring-boot-ionic-example). It also expects your apps to be named ionic-beer and server. If you don’t have a Cloud Foundry account, you’ll need to create one. Then install its command line tools (and login) for this script to work. brew tap cloudfoundry/tap && brew install cf-cli cf login -a api.run.pivotal.io After logging into Cloud Foundry, you should be able to run ./deploy.sh. This script will deploy the server and modify beer.service.ts to point to it. It will also try to deploy to your phone, so you may need to complete the steps below before it will work. To deploy the app to an iPhone, start by plugging it into your computer. Then run the following commands to install ios-deploy/ios-sim, build the app, and run it on your device. npm install -g ios-deploy ios-sim ionic cordova build ios --prod open platforms/ios/ionic-beer.xcodeproj Select your phone as the target in Xcode and click the play button to run your app. The first time you do this, Xcode may spin for a while with a “Processing symbol files” message at the top. NOTE: If you run into code signing issues, see Ionic’s deployment documentation to see how to solve. Once you’ve configured your phone, computer, and Apple ID to work, you should be able to open the app and see the beer list you created. Below is how it looks on my iPhone 7 Plus. Android To emulate or deploy to an Android device, you’ll first need to install Android Studio. As part of the install, it will show you where it installed the Android SDK. Set this path as an ANDROID_HOME environment variable. On a Mac, it should be ~/Library/Android/sdk/. If you’ve just installed Android Studio, make sure to open it to complete the installation. To deploy to the Android emulator, run ionic cordova emulate android. This will install Android support and display an error if you don’t have any AVD (Android Virtual Device) images. (node:9300) UnhandledPromiseRejectionWarning: CordovaError: No emulator images (avds) found. 1. Download desired System Image by running: /Users/mraible/Library/Android/sdk/tools/android sdk 2. Create an AVD by running: /Users/mraible/Library/Android/sdk/tools/android avd HINT: For a faster emulator, use an Intel System Image and install the HAXM device driver To create a new AVD, open Android Studio and navigate to Tools > Android > AVD Manager. Create a new Virtual Device and click Play. I chose a Pixel 2. After performing these steps, you should be able to run ionic cordova emulate android and see your app running in the AVD. Learn More about Ionic and Angular I hope you’ve enjoyed this tour of Ionic and Angular. I like how Ionic takes your web development skills up a notch and allows you to create mobile applications that look and behave natively. You can find a completed version of the application created in this blog post on GitHub. If you encountered issues, please create an issue in GitHub or hit me up on Twitter @mraible. To learn more about Ionic and Angular, please see the following resources: Update: To learn how to add authentication to an Ionic app, see Build an Ionic App with User Authentication. Page 2 You’re developing a Progressive Web Application (PWA), and your service worker and web app manifest are working swimmingly. You’ve even taken the time to deploy it to a server with HTTPS, and you’re feeling pretty good about things. But wait, you don’t have any way of knowing who your users are! Don’t you want to provide them with an opportunity to authenticate and tell you who they are? Once you know who they are, you can give them all kinds of personalization options, inspire them to ❤ your app, and maybe even support your work! In this article, I’ll show you how you can lock down a Spring Boot app, then use a modern authentication protocol, in this case, OpenID Connect (OIDC), to authenticate and gain access to its APIs. Secure Your Spring Boot App You might’ve heard that Stormpath joined forces with Okta a few months ago (February 2017). Since the transition, we’ve been working hard to make the Stormpath SDKs work with Okta’s API. The good news is we’ve made significant progress! In this example, you’ll use Okta’s Spring Boot Starter to add security to a Spring Boot app. Then I’ll show you how you can use OIDC and Okta’s Angular SDK in an Angular app to log in and get data from the Spring Boot app. I recently created a Spring Boot app that provides a list of good beers, based on a pre-populated list. It filters out less-than-great beers and displays them in an Angular UI that displays the first animated GIF (from Giphy) that matches the beer name. Let’s get started! Rather than building Spring Boot and Angular applications from scratch, you can clone an existing GitHub project to get you going quickly. git clone If you’d prefer to build this application yourself, please read Build Your First Progressive Web Application with Angular and Spring Boot. In this project’s server/pom.xml file, you’ll need to add the Okta Spring Boot starter as a dependency, and force Spring Security to use the latest version of Spring Security OAuth: 0.3.0 ... com.okta.spring okta-spring-boot-starter ${okta.version} org.springframework.security.oauth spring-security-oauth2 2.2.1.RELEASE NOTE: You’ll need to comment out the devtools dependency to work around an issue in the Okta Spring Boot Starter. Get Started with Okta To begin, you’ll need to create an Okta Developer account. This account is free forever and provides the complete Okta Identity Platform for up to 3 applications and 100 users. Head on over to Fill out the signup form, and click “Get Started” Within a few minutes, you’ll get a confirmation email, follow the instructions in the email to finish setting up your account Log in to your Okta Developer account and navigate to Applications > Add Application. Click Single-Page App, click Next, and give the app a name you’ll remember (e.g., “Angular PWA”). Change all instances of localhost:8080 to localhost:4200 and click Done. TIP: Add as a Logout redirect URI, so log out functionality works in your Angular app. Copy the client ID into your server/src/main/resources/application.properties file. While you’re in there, add a okta.oauth2.issuer property that matches your Okta domain. For example: okta.oauth2.issuer=https://{yourOktaDomain}/oauth2/default okta.oauth2.clientId={clientId} Update server/src/main/java/com/okta/developer/demo/DemoApplication.java to enable it as a resource server. import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer; @EnableResourceServer @SpringBootApplication Now, start it up… cd server ./mvnw spring-boot:run Navigate to , and you’ll see an access denied error. If you call the API with a different Accept header (e.g. application/json), you’ll get a JSON response. The command below uses HTTPie. $ http :8080 HTTP/1.1 401 Cache-Control: no-cache, no-store, max-age=0, must-revalidate Cache-Control: no-store Content-Type: application/json;charset=UTF-8 Date: Fri, 23 Feb 2018 18:50:58 GMT Expires: 0 Pragma: no-cache Pragma: no-cache Transfer-Encoding: chunked WWW-Authenticate: Bearer realm="api://default", error="unauthorized", error_description="Full authentication is required to access this resource" X-Content-Type-Options: nosniff X-Frame-Options: DENY X-XSS-Protection: 1; mode=block { "error": "unauthorized", "error_description": "Full authentication is required to access this resource" } Authenticate with OpenID Connect Start the Angular application by running the following commands in your project’s root directory. cd client npm install ng serve If you receive an error like the one below, disable the warning using the instructions provided, or just ignore

it. You can upgrade your dependencies in package.json, but there’s no guarantee that this tutorial will still work! ;) Your global Angular CLI version (1.7.0) is greater than your local version (1.6.7). The local Angular CLI version is used. To disable this warning use "ng set --global warnings.versionMismatch=false". When you navigate to , you’ll likely see a cross-origin request error. Failed to load No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin ' ' is therefore not allowed access. The response had HTTP status code 401. You can usually use a @CrossOrigin annotation to enable cross-origin resource sharing (CORS) on the server, but it won’t work once you integrate Spring Security. To solve this issue, create a simpleCorsFilter bean in your DemoApplication.java class. package com.example.demo; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.boot.web.servlet.FilterRegistrationBean; import org.springframework.context.annotation.Bean; import org.springframework.core.Ordered; import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer; import org.springframework.web.cors.CorsConfiguration; import org.springframework.web.cors.UrlBasedCorsConfigurationSource; import org.springframework.web.filter.CorsFilter; import java.util.Collections; @EnableResourceServer @SpringBootApplication public class DemoApplication { public static void main(String[] args) { SpringApplication.run(DemoApplication.class, args); } @Bean public FilterRegistrationBean simpleCorsFilter() { UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource(); CorsConfiguration config = new CorsConfiguration(); config.setAllowCredentials(true); config.setAllowedOrigins(Collections.singletonList(" ")); config.setAllowedMethods(Collections.singletonList("*")); config.setAllowedHeaders(Collections.singletonList("*")); source.registerCorsConfiguration("/**", config); FilterRegistrationBean bean = new FilterRegistrationBean(new CorsFilter(source)); bean.setOrder(Ordered.HIGHEST_PRECEDENCE); return bean; } } You can remove the @CrossOrigin annotation from BeerController.java since it’s no longer needed. Make sure to save the files you changed and restart your server. Install Manfred Steyer’s project to add OAuth 2 and OpenID Connect support to your Angular client. npm install --save angular-oauth2-oidc Update client/src/app/app.component.ts to import OAuthService and configure your app to use your Okta application settings (replacing {clientId} and {yourOktaDomain} with the values from your “Angular PWA” OIDC app). import { JwksValidationHandler, OAuthService } from 'angular-oauth2-oidc'; ... constructor(private oauthService: OAuthService) { this.oauthService.redirectUri = window.location.origin; this.oauthService.clientId = '{clientId}'; this.oauthService.scope = 'openid profile email'; this.oauthService.oidc = true; this.oauthService.issuer = 'https://{yourOktaDomain}/oauth2/default'; this.oauthService.tokenValidationHandler = new JwksValidationHandler(); this.oauthService.loadDiscoveryDocumentAndTryLogin(); } ... Modify client/src/app/app.component.html to use instead of . Create client/src/app/home.component.ts and configure it to display Login and Logout buttons. import { Component } from '@angular/core'; import { OAuthService } from 'angular-oauth2-oidc'; @Component({ template: ` Welcome, {{givenName}}! Logout Beer List Login ` }) export class HomeComponent { constructor(private oauthService: OAuthService) { } login() { this.oauthService.initImplicitFlow(); } logout() { this.oauthService.logOut(); } get givenName() { const claims: any = this.oauthService.getIdentityClaims(); if (!claims) { return null; } return claims.name; } } Modify client/src/app/shared/beer/beer.service.ts to read the access token from oauthService and add an Authorization header. import { Injectable } from '@angular/core'; import { HttpClient, HttpHeaders } from '@angular/common/http'; import { Observable } from 'rxjs/Observable'; import { OAuthService } from 'angular-oauth2-oidc'; @Injectable() export class BeerService { constructor(private http: HttpClient, private oauthService: OAuthService) {} getAll(): Observable { return this.http.get(' , {headers: this.getHeaders()}); } getHeaders(): HttpHeaders { return new HttpHeaders().set('Authorization', this.oauthService.authorizationHeader()); } } Create client/src/app/shared/auth/auth.guard.ts to navigate to the HomeComponent if the user is not authenticated. import { Injectable } from '@angular/core'; import { ActivatedRouteSnapshot, CanActivate, Router, RouterStateSnapshot } from '@angular/router'; import { OAuthService } from 'angular-oauth2-oidc'; @Injectable() export class AuthGuard implements CanActivate { constructor(private oauthService: OAuthService, private router: Router) {} canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean { if (this.oauthService.hasValidIdToken()) { return true; } this.router.navigate(['/home']); return false; } } Add OAuthModule.forRoot() to the list of imports in client/src/app/app.module.ts, add HomeComponent in declarations, and lock the /beer-list route down with the AuthGuard. While you’re there, add MatCardModule to the list of imports since HomeComponent uses components from this module. import { OAuthModule } from 'angular-oauth2-oidc'; import { HomeComponent } from './home/home.component'; import { RouterModule, Routes } from '@angular/router'; import { AuthGuard } from './shared/auth/auth.guard'; import { MatCardModule } from '@angular/material'; const appRoutes: Routes = [{ path: 'beer-list', component: BeerListComponent, canActivate: [AuthGuard] }, { path: 'home', component: HomeComponent }, { path: '', redirectTo: 'home', pathMatch: 'full' }, { path: '**', redirectTo: 'home' }]; @NgModule({ declarations: [... HomeComponent], imports: [... MatCardModule, OAuthModule.forRoot(), RouterModule.forRoot(appRoutes)], providers: [AuthGuard], bootstrap: [AppComponent] }) export class AppModule { } After making these changes, you should be able to run ng serve and see a login button. Click the Login button and sign-in with one of the users assigned in your Okta application. You should see a welcome message like the one below. Click on Beer List to see data from your Spring Boot app. To add the “Home” link at the top (as shown in the screenshot above), modify client/src/app/beer-list/beer-list.component.html to include the following HTML. Home Beer List

{{beer.name}}

If it works - great, now we can add auth with Okta! Authenticating with the Okta Auth SDK The Okta Auth SDK builds on top of Okta’s Authentication API and OAuth 2.0 API to enable you to create a fully branded sign-in experience using JavaScript. Install it using npm: npm install @okta/okta-auth-js --save Change HomeComponent to import OktaAuth and modify its template, so it has a sign-in form. import * as OktaAuth from '@okta/okta-auth-js'; @Component({ template: ` Welcome, {{givenName}}! Logout Beer List Login with Redirect Login Login Directly {{error}} Login ` }) Update client/src/app/app.module.ts to import the dependencies required by Angular Material when working with forms. import { MatInputModule } from '@angular/material'; import { FormsModule } from '@angular/forms'; import { BrowserAnimationsModule } from '@angular/platform-browser/animations'; @NgModule({ ... imports: [... FormsModule, BrowserAnimationsModule, MatInputModule], ... }) After making these changes, run ng serve and the HomeComponent should render as follows (after you’ve logged out and unchecked “Offline” in the Network tab): In HomeComponent, import Angular’s ChangeDetectorRef, add it as a dependency in the constructor, and add local variables for the username and password fields. Then implement a loginWithPassword() method in HomeComponent. This method uses the OktaAuth library to get a session token and exchange it for ID and access tokens. import { ChangeDetectorRef } from '@angular/core'; ... export class HomeComponent { private username: string; private password: string; private error: string; constructor(private oauthService: OAuthService, private changeDetector: ChangeDetectorRef) { } ... loginWithPassword() { this.oauthService.createAndSaveNonce().then(nonce => { const authClient = new OktaAuth({ url: 'https://{yourOktaDomain}', issuer: 'default' }); return authClient.signIn({ username: this.username, password: this.password }).then((response) => { if (response.status === 'SUCCESS') { return authClient.token.getWithoutPrompt({ clientId: this.oauthService.clientId, responseType: ['id_token', 'token'], scopes: ['openid', 'profile', 'email'], sessionToken: response.sessionToken, nonce: nonce, redirectUri: window.location.origin }) .then((tokens) => { const idToken = tokens[0].idToken; const accessToken = tokens[1].accessToken; const keyValuePair = `#id_token=${encodeURIComponent(idToken)}&access_token=${encodeURIComponent(accessToken)}`; this.oauthService.tryLogin({ customHashFragment: keyValuePair, disableOAuth2StateCheck: true }).then() => { // notify Angular that things have changed this.changeDetector.detectChanges(); }); }); } else { throw new Error('We cannot handle the ' + response.status + ' status'); } }).fail((error) => { console.error(error); this.error = error.message; }); }); } } You should be able to sign in with the form as one of your app’s registered users. After logging in, you’ll be able to click the Beer List link and view the beer list. Deploy to Cloud Foundry Now it’s time for one of the coolest places on the internet - production! You’ll need to modify the deploy.sh script in the root directory to replace in DemoApplication.java instead of BeerController.java, but that’s about it. You can see the modified deploy.sh on GitHub. Copy the contents of this file on top of your existing deploy.sh. Install the Cloud Foundry CLI, then log into Pivotal Web Services. cf login -a api.run.pivotal.io Run ./deploy.sh and watch the magic happen! If you navigate to the client’s URL after deploying, you’ll see an error like the following in Chrome’s console. Failed to load No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin ' ' is therefore not allowed access. To fix this, modify the Trusted Origins on Okta (under API > Trusted Origins) to have your client’s URL (e.g.). You’ll also need to add this URL to the Login redirect URI and Logout redirect URI properties for your “Angular PWA” OIDC application. After making these changes, both login techniques should work as expected and you should be able to load the beer list from your Spring Boot app. After I got everything deployed, I used Lighthouse to audit this application and found it received a perfect score. Huzzah! TIP: Joe Kutner, Java Languages Owner at Heroku, created a heroku.sh script you can use to deploy this same application to Heroku. Thanks, Joe! Happy Authenticating! You can find the source code associated with this article on GitHub. If you find any bugs, please file an issue, or post your question to the Okta Developer Forums. Of course, you can always ping me on Twitter too. This article showed you how to add authentication with Okta to an Angular PWA with a Spring Boot API. You learned how to use OpenID Connect to get an access token and securely communicate with the backend. Finally, you saw how to deploy everything to Cloud Foundry and get a Lighthouse PWA score of 100. To learn more about PWAs, check out some recent tutorials I wrote: There’s also a number of excellent resources by Google and Smashing Magazine: Page 3 Adopting a microservice architecture provides unique opportunities to add failover and resiliency to your systems, so your components can handle load spikes and errors gracefully. Microservices make change less expensive too. It can also be a good idea when you have a large team working on a single product. Your project can likely be broken up into components that can function independently of one another. Once components can function independently, they can be built, tested, and deployed independently. This gives an organization and its teams the agility to develop and deploy very quickly. In a previous article, I showed you how to build a Spring Boot API with an Angular client. I then showed you how to convert the Angular app into a progressive web application that works offline. The Angular PWA is a good example of a resilient application because it still works when connectivity fails. Did you know you can develop similar resiliency in your API with Spring Boot, Spring Cloud, and a microservices architecture? This article shows you how to convert the previously created Spring Boot application to use microservices. You’ll create a beer catalog service, an edge service (for filter and displaying good beers), and a Eureka service that registers the services and allows them to communicate with one another. Before we dive into the code tutorial, I’d like to talk a bit about microservices, their history, and why you should (or should not) consider a microservices architecture for your next project. History of Microservices According to Wikipedia, the term “microservice” was first used as a common architecture style at a workshop of software architects near Venice in May 2011. In May 2012, the same group decided “microservices” was a more appropriate name. Adrian Cockcroft, who was at Netflix at the time, described this architecture as “fine-grained SOA”. Martin Fowler and James Lewis wrote an article titled simply Microservices on March 25, 2014. Years later, this is still considered the definitive article for microservices. Organizations and Conway’s Law Technology has traditionally been organized into technology layers: UI team, database team, operations team. When teams are separated along these lines, even simple changes can lead to a cross-team project sucking down huge chunks of time and budget. A smart team will optimize around this and choose the lesser of two evils; forcing the logic into whichever application they have access to. This is an example of Conway’s Law in action. Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization’s communication structure. — Melvyn Conway, 1967 Microservices Architecture Philosophy The philosophy of a microservices architecture is essentially equal to the Unix philosophy of “Do one thing and do it well”. The characteristics of a microservices architecture are as follows: Componentization via services Organized around business capabilities Products not projects Smart endpoints and dumb pipes Decentralized governance Decentralized data management Infrastructure automation Design for failure Evolutionary design Why Microservices? For most developers, dev teams, and organizations, it’s easier to work on small “do one thing well” services. No single program represents the whole application, so services can change frameworks (or even languages) without a massive cost. As long as the services use a language agnostic protocol (HTTP or lightweight messaging), applications can be written in several different platforms - Java, Ruby, Node, Go, .NET, etc. - without issue. Platform-as-a-Service (PaaS) providers and containers have made it easy to deploy microservices. All the technologies needed to support a monolith (e.g. load balancing, discovery, process monitoring) are provided by the PaaS, outside of your container. Deployment effort comes close to zero. Are Microservices the Future? Architecture decisions, like adopting microservices, are usually only evident several years after you make them. Microservices have been successful at companies like LinkedIn, Twitter, Facebook, Amazon, and Netflix. But that doesn’t mean they’ll be successful for your organization. Component boundaries are hard to define. If you’re not able to create your components cleanly, you’re just shifting complexity from inside a component to the connections between the components. Also, team capabilities are something to consider. A weak team will always create a weak system. You shouldn’t start with a microservices architecture. Instead, begin with a monolith, keep it modular, and split it into microservices once the monolith becomes a problem. — Martin Fowler Build a Microservices Architecture with Spring Boot, Spring Cloud, and Netflix Eureka Netflix Eureka is a REST-based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers. Spring Cloud is a developer’s dream when it comes to implementing and deploying a microservices architecture. Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, etc.). Coordination of distributed systems leads to boilerplate patterns. Using Spring Cloud, developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer’s own laptop, bare metal data centers, and managed platforms such as Cloud Foundry. Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot applications. Patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon). To learn more about service discovery and resolution with Eureka, watch Josh Long’s Microservice Registration and Discovery with Spring Cloud and Netflix’s Eureka. Create a Eureka Service To begin, create a spring-boot-microservices-example directory on your hard drive. Navigate to start.spring.io. Enter eureka-service as an artifact name and select Eureka Server as a dependency. Click the Generate Project button and expand eureka-service.zip into the spring-boot-microservices-example directory. TIP: You could also create your project using start.spring.io’s API. The following HTTPie command will create the same app as the steps above: http artifactId==eureka-service bootVersion==2.0.5.RELEASE \ name==eureka-service dependencies==cloud-eureka-server baseDir==eureka-service | tar -xzvf - Modify eureka-service/src/main/resources/application.properties to add a port number and disable registration. server.port=8761 eureka.client.register-with-eureka=false Open eureka-service/src/main/java/com/example/eurekaservice/EurekaServiceApplication.java and add @EnableEurekaServer above @SpringBootApplication. import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer; @EnableEurekaServer @SpringBootApplication This annotation, and the aforementioned properties, configures a registry that other applications can talk to. Start the application from the command line using: Or if you’re using Windows: After it starts, you should be able to open and see there are no services available. Create a Beer Catalog Service Refresh start.spring.io to start creating a new project. Use beer-catalog-service for the artifact name and add the following dependencies: Actuator: features to help you monitor and manage your application Eureka Discovery: for service registration JPA: to save/retrieve data H2: an in-memory database Rest Repositories: to expose JPA repositories as REST endpoints Web: Spring MVC and embedded Tomcat DevTools: to auto-reload the application when files change Lombok: to reduce boilerplate code Click the Generate Project button and expand beer-catalog-service.zip into spring-boot-microservices-example and open the project in your favorite IDE. I recommend IntelliJ IDEA because it’s great for Java and web development. TIP: To create this same project using start.spring.io’s API, run the following: http artifactId==beer-catalog-service bootVersion==2.0.5.RELEASE \ name==beer-catalog-service dependencies==actuator,cloud-eureka,data-jpa,h2,data-rest,web,devtools,lombok \ baseDir==beer-catalog-service | tar -xzvf - Create a Beer entity, a JpaRepository for it, and a CommandLineRunner to populate the database with default data. You can add this code to BeerCatalogServiceApplication.java, or create separate files for each class. The code below assumes you’re putting all classes in the same file. @Data @AllArgsConstructor @Entity class Beer { public Beer(String name) { this.name = name; } @Id @GeneratedValue private Long id; private String name; } @RepositoryRestResource interface BeerRepository extends JpaRepository {} @Component class BeerInitializer implements CommandLineRunner { private final BeerRepository beerRepository; BeerInitializer(BeerRepository beerRepository) { this.beerRepository = beerRepository; } @Override public void run(String... args) throws Exception { Stream.of("Kentucky Brunch Brand Stout", "Good Morning", "Very Hazy", "King Julius", "Budweiser", "Coors Light", "PBR") .forEach(beer -> beerRepository.save(new Beer(beer))); beerRepository.findAll().forEach(System.out::println); } } If you’re using an editor that doesn’t auto-import classes, here’s the list of imports needed at the top of BeerCatalogServiceApplication.java. import lombok.AllArgsConstructor; import lombok.Data; import org.springframework.boot.CommandLineRunner; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.client.discovery.EnableDiscoveryClient; import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.data.rest.core.annotation.RepositoryRestResource; import org.springframework.stereotype.Component; import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.Id; import java.util.stream.Stream; Add an application name in beer-catalog-service/src/main/resources/application.properties to display in the Eureka service, and set the port to 8080. server.port=8080 spring.application.name=beer-catalog-service Start the beer-catalog-service with Maven (mvn spring-boot:run) or your IDE. At this point, you should be able to use HTTPie to see the list of beers from the catalog service. However, if you open the Eureka Service at , you will not see the service registered. To register the beer-catalog-service, you need to add @EnableDiscoveryClient to BeerCatalogServiceApplication.java. import org.springframework.cloud.client.discovery.EnableDiscoveryClient; @EnableDiscoveryClient @SpringBootApplication public class BeerCatalogServiceApplication { public static void main(String[] args) { SpringApplication.run(BeerCatalogServiceApplication.class, args); } } Re-compile this class, watch devtools restart your application, and return to . If you’re not using an IDE, it might be easiest to cancel and restart mvn spring-boot:run. Now the service should show up. Compile on Save in IntelliJ By default IntelliJ IDEA does not automatically compile files when the application is running. To enable the "Compile on save" feature: Go to Preferences > Build, Execution, Deployment -> Compiler and enable "Build project automatically" Open the Action window: Linux: CTRL+SHIFT+A Mac: SHIFT+COMMAND+A Windows: CTRL+ALT+SHIFT+/ Enter Registry... and enable compiler.automake.allow.when.app.running Create an Edge Service The edge service will be similar to the standalone beer service created in Bootiful Development with Spring Boot and Angular. However, it will have fallback capabilities which prevent the client from receiving an HTTP error when the service is not available. Navigate to start.spring.io and create an edge-service application with the following dependencies: Eureka Discovery: for service registration Feign: a declarative web service client Zuul: provides intelligent routing Rest Repositories: to expose JPA repositories as REST endpoints Web: Spring MVC and embedded Tomcat Hystrix: a circuit breaker to stop cascading failure and enable resilience Lombok: to reduce boilerplate code Click the Generate Project button and expand edge-service.zip into spring-boot-microservices-example and open the project in your favorite IDE. TIP: To create this same project using start.spring.io’s API, run the following: http artifactId==edge-service bootVersion==2.0.5.RELEASE \ name==edge-service dependencies==cloud-eureka,cloud-feign,cloud-zuul,data-rest,web,cloud-hystrix,lombok \ baseDir==edge-service | tar -xzvf - Since the beer-catalog-service is running on port 8080, you’ll need to configure this application to run on a different port. Modify edge-service/src/main/resources/application.properties to set the port to 8081 and set an application name. server.port=8081 spring.application.name=edge-service To enable Feign, Hystrix, and registration with the Eureka server, add the appropriate annotations to EdgeServiceApplication.java: package com.example.edgeservice; import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand; import lombok.Data; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker; import org.springframework.cloud.client.discovery.EnableDiscoveryClient; import org.springframework.cloud.netflix.feign.EnableFeignClients; import org.springframework.cloud.netflix.feign.FeignClient; import org.springframework.cloud.netflix.zuul.EnableZuulProxy; import org.springframework.hateoas.Resources; import org.springframework.web.bind.annotation.*; import java.util.ArrayList; import java.util.Collection; import java.util.stream.Collectors; @EnableFeignClients @EnableCircuitBreaker @EnableDiscoveryClient @EnableZuulProxy @SpringBootApplication public class EdgeServiceApplication { public static void main(String[] args) { SpringApplication.run(EdgeServiceApplication.class, args); } } Create a Beer DTO (Data Transfer Object) in this same file. Lombok’s @Data will generate a toString() methods, getters, setters, and appropriate constructors. @Data class Beer { private String name; } Create a BeerClient interface that uses Feign to talk to the beer-catalog-service. public class EdgeServiceApplication { public static void main(String[] args) { SpringApplication.run(EdgeServiceApplication.class, args); } } @Data class Beer { private String name; } @FeignClient("beer-catalog-service") interface BeerClient { @GetMapping("/beers") Resources readBeers(); } Create a RestController below the BeerClient that filters out less-than-great beers and exposes a /good-beers endpoint. NOTE: To get beer.getName() to work in your IDE, you may need to install the Lombok plugin. In Intellij, you can install it by going to Preferences -> Plugins -> Browse Plugins. Search for “lombok plugin” and click to install it. Restart Intellij for the changes to take effect. @RestController class GoodBeerApiAdapterRestController { private final BeerClient beerClient; public GoodBeerApiAdapterRestController(BeerClient beerClient) { this.beerClient = beerClient; } @GetMapping("/good-beers") public Collection goodBeers() { return beerClient.readBeers() .getContent() .stream() .filter(this::isGreat) .collect(Collectors.toList()); } private boolean isGreat(Beer beer) { return !beer.getName().equals("Budweiser") && !beer.getName().equals("Coors Light") && !beer.getName().equals("PBR"); } } Start the edge-service application with Maven or your IDE and verify it registers successfully with the Eureka server. You should be able to invoke the /good-beers endpoint as well. $ http :8081/good-beers HTTP/1.1 200 Content-Type: application/json;charset=UTF-8 Date: Fri, 11 May 2018 17:28:55 GMT Transfer-Encoding: chunked [{ "name": "Kentucky Brunch Brand Stout" }, { "name": "Good Morning" }, { "name": "Very Hazy" }, { "name": "King Julius" }] This is cool, but if you shut down the beer-catalog-service application, you’ll get a 500 internal server error. $ http :8081/good-beers HTTP/1.1 500 Connection: close Content-Type: application/json;charset=UTF-8 Date: Fri, 11 May 2018 17:35:39 GMT Transfer-Encoding: chunked { "error": "Internal Server Error", "message": "connect timed out executing GET , "path": "/good-beers", "status": 500, "timestamp": "2018-05-11T17:35:39.201+0000" } To fix this, you can use Hystrix to create a fallback method and tell the goodBeers() method to use it. public Collection fallback() { return new ArrayList(); } @HystrixCommand(fallbackMethod = "fallback") @GetMapping("/good-beers") public Collection goodBeers() { return ... } Restart the edge-service and you should see an empty list returned. $ http :8081/good-beers HTTP/1.1 200 Content-Type: application/json;charset=UTF-8 Date: Fri, 11 May 2018 17:38:18 GMT Transfer-Encoding: chunked Start the beer-catalog-service again and this list should eventually return the full list of good beer names. Add an Angular PWA Client You can copy the Angular PWA client I created in a previous tutorial and install its dependencies. git clone cp -r spring-boot-angular-pwa-example/client ~/spring-boot-microservices-example/. cd ~/spring-boot-microservices-example/client npm install Then modify the BeerService in client/src/app/shared/beer/beer.service.ts to use port 8081 instead of 8080. getAll(): Observable { return this.http.get('); } Modify GoodBeerApiAdapterRestController in EdgeServiceApplication.java to allow cross-origin requests from any client. @GetMapping("/good-beers") @CrossOrigin(origins = "*") public Collection goodBeers() { Restart the edge-service and start the Angular client by running npm start in the client directory. Open in your browser and verify that network calls to /good-beers go over port 8081. Deploy to Cloud Foundry In order to deploy the edge-service and beer-catalog-service to Cloud Foundry, you need to add configuration files so they work with Cloud Foundry’s Eureka service. Create edge-service/src/main/resources/application-cloud.properties and populate it with the following: eureka.instance.hostname=${vcap.application.uris[0]:localhost} eureka.instance.nonSecurePort=80 eureka.instance.metadataMap.instanceId=${vcap.application.instance_id:${spring.application.name}:${spring.application.instance_id:${server.port}}} eureka.instance.leaseRenewalIntervalInSeconds = 5 eureka.client.region = default eureka.client.registryFetchIntervalSeconds = 5 eureka.client.serviceUrl.defaultZone=${vcap.services.pwa-eureka-service.credentials.uri}/eureka/ Create beer-catalog-service/src/main/resources/application-cloud.properties and populate it with similar properties. eureka.instance.hostname=${vcap.application.uris[0]:localhost} eureka.instance.nonSecurePort=80 eureka.instance.metadataMap.instanceId=${vcap.application.instance_id:${spring.application.name}:${spring.application.instance_id:${server.port}}} eureka.instance.leaseRenewalIntervalInSeconds = 5 eureka.client.region = default eureka.client.registryFetchIntervalSeconds = 5 eureka.client.serviceUrl.defaultZone=${vcap.services.pwa-eureka-service.credentials.uri}/eureka/ In the properties above, pwa-eureka-service is the name you’ll give to the Eureka service when you deploy it to Cloud Foundry. To deploy it on Cloud Foundry with Pivotal Web Services, you’ll need to create an account, download/install the Cloud Foundry CLI, and sign-in (using cf login -a api.run.pivotal.io). There are quite a few steps involved to deploy all the services and the Angular client for production. For that reason, I wrote a deploy.sh script that automates everything. TIP: If you receive an error stating that you’re using too much memory, you may have to upgrade your Cloud Foundry subscription. When to Use Microservices Building a microservices architecture is something you should consider when you’re having difficulty scaling development in a large team. From a development standpoint, moving to microservices will not reduce complexity, but will likely increase it you move to a distributed system. Automation and orchestration are key for deployment. You should make sure to define your exit criteria (e.g. maximum time for a request to execute) before implementing your microservices infrastructure. You’re likely going to have to custom build some things, so be prepared for that. Trial a few different platforms and then pick the one that meets your criteria and is the easiest to develop with. Don’t develop half of your system on one platform and then try moving to another. Another tip is to make sure and record the request ID in all logging events for traceability. If you have fewer than 20 developers, start with a monolith, but build in async messaging as soon as possible. Use it for things like mail, notifications, logging, and archiving. Debugging, deployment, and logging are much easier with a monolith because everything is contained in one application. Also, consider using async messaging or other non-blocking communication methods with automatic back pressure. HTTP is a synchronous protocol and can be a limiting factor in high-traffic systems. Learn More about Microservice Architectures Spring Boot isn’t the only framework to implement embedded servlet containers or make it easy to develop microservices. In Javaland, there’s Dropwizard, MicroProfile for Java EE, Lagom, and Vert.x, and Tribestream. You can find the source code for this article’s applications on GitHub at . You can also watch a video of me and Josh Long developing these applications in a YouTube recording of our Cloud Native PWAs presentation at Devoxx France, 2017. If you have any questions about this article, you can email me at matt.raible@okta.com, post a question to Stack Overflow with the Okta tag, post to our Developer Forums, or create an issue on GitHub. Update: To learn about how security fits into all this, see Secure a Spring Microservices Architecture with Spring Security, JWTs, Juiser, and Okta. Update 2: To learn how to lock this application down with Spring Security and OAuth, see Secure a Spring Microservices Architecture with Spring Security and OAuth 2.0. Page 4 JHipster is one of those open-source projects you stumble upon and immediately think, “Of course!” It combines three very successful frameworks in web development: Bootstrap, Angular, and Spring Boot. Bootstrap was one of the first dominant web-component frameworks. Its largest appeal was that it only required a bit of HTML and it worked! Bootstrap showed many in the Java community how to develop components for the web. It leveled the playing field in HTML/CSS development, much like Apple’s Human Interface Guidelines did for iOS apps. At its core, JHipster is a Yeoman generator. Yeoman is a code generator that you run with a yo command to generate complete applications or useful pieces of an application. Yeoman generators promote what the Yeoman team calls the “Yeoman workflow”. This is an opinionated client-side stack of tools that can help developers quickly build beautiful web applications. It takes care of providing everything needed to get working without the normal pains associated with a manual setup. Learn more about JHipster, including its origin, at . This tutorial shows you how to build a microservices architecture with JHipster 4.5.4. You’ll generate a gateway (powered by Netflix Zuul and the JHipster Gateway), a microservice (that talks to MongoDB), and use Docker Compose to make sure it all runs locally. Then you’ll deploy it to Minikube and Google Cloud using Kubernetes. Install JHipster 4 The Installing JHipster instructions show you all the tools you’ll need to use a released version of JHipster. Microservices with JHipster To build a microservices architecture with JHipster, you’ll need to generate two applications and clone another. Generate a gateway Generate a microservice Clone the JHipster Registry You can see how these components fit in the diagram below. To see what’s happening inside your applications, you can use the JHipster Console, a monitoring tool based on the ELK Stack. I’ll cover JHipster Console in the Docker Compose section. Create an API Gateway To create a microservices project, open a terminal window and create a jhipster-microservices-example directory. Then create a blog directory for the gateway application. mkdir -p jhipster-microservices-example/blog In JHipster terms, a gateway is a normal JHipster application. This means you can develop it like a monolith, but it also acts as the entrance to your microservices. More specifically, it provides HTTP routing and load balancing, quality of service, security, and API documentation for all microservices. In a terminal, navigate to the blog directory and run jhipster. cd jhipster-microservices-example/blog jhipster JHipster prompts you with many questions about the type of application you want to generate and what features you’d like to include. Create the blog application with the following settings: Application type: Microservice gateway Base name of the application: blog Port: 8080 Default package name: org.jhipster.blog JHipster Registry: Yes Type of authentication: JWT Type of database: SQL Production database: PostgreSQL Development database: H2 with disk-based persistence Maven or Gradle: Maven Other technologies: Elasticsearch Client framework: Angular 4 Sass for CSS: Yes Internationalization support: Yes Native language: English Additional languages: Spanish Testing frameworks: Gatling, Protractor Install other generators from the JHipster Marketplace: No If you’d like to use the same settings I did, you can place the following .yo-rc.json file in the blog directory and run jhipster in it. You won’t be prompted to answer any questions because the answers are already in .yo-rc.json. { "generator-jhipster": { "promptValues": { "packageName": "org.jhipster.blog", "nativeLanguage": "en" }, "jhipsterVersion": "4.5.4", "baseName": "blog", "packageName": "org.jhipster.blog", "packageFolder": "org/jhipster/blog", "serverPort": "8080", "authenticationType": "jwt", "hibernateCache": "hazelcast", "clusteredHttpSession": false, "websocket": false, "databaseType": "sql", "devDatabaseType": "h2Disk", "prodDatabaseType": "postgresql", "searchEngine": "elasticsearch", "messageBroker": false, "serviceDiscoveryType": "eureka", "buildTool": "maven", "enableSocialSignIn": false, "jwtSecretKey": "f7450a034a7251b3d201cf5139852f9adf69c2d1", "clientFramework": "angular2", "useSass": true, "clientPackageManager": "yarn", "applicationType": "gateway", "testFrameworks": ["gatling", "protractor"], "jhiPrefix": "jhi", "enableTranslation": true, "nativeLanguage": "en", "languages": ["en", "es"] } } The project creation process will take a couple of minutes to run, depending on your internet connection speed. When it’s finished, you should see output like this: Before you can run this project, you’ll need to download and start an instance of the JHipster Registry. Run the following commands in the jhipster-microservices-example directory. git clone git@github.com:jhipster/jhipster-registry.git registry cd registry && yarn && ./mvnw The JHipster Registry is built on Spring Cloud Netflix and Spring Cloud Config. Patterns provided by Spring Cloud Netflix include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul), and Client Side Load Balancing (Ribbon). In a previous post, I showed you how you can use Eureka for service discovery. JHipster Registry is a Eureka server, a Spring Cloud Config server, as well as an administration server. It includes dashboards to monitor and manage your JHipster applications. JHipster Registry starts on port 8761 by default. In a new terminal window, navigate to jhipster-microservices-example/blog and run ./mvnw to start the blog application and open in your favorite browser. The first thing you’ll notice is a dapper-looking fellow explaining how you can sign in or register. Sign in with username admin and password admin and you’ll have access to navigate through the Administration section. This section offers nice looking UIs on top of some Spring Boot’s many monitoring and configuration features. It also allows you to administer users: It gives you insights into Application and JVM metrics: And it allows you to see the Swagger docs associated with its API. You can run the following command (in a separate terminal window) to start the Protractor tests and confirm everything is working properly. At this point, it’s a good idea to check your project into Git so you can easily see what changes are made going forward. git init git add . git commit -m "Gateway created" Shut down your blog application before proceeding to the next section. Generate Entities For each entity you want to create, you will need: A database table A Liquibase change set A JPA entity class A Spring Data JpaRepository interface A Spring MVC RestController class An Angular model, state, component, dialog components, service Several HTML pages for each component Also, you should have integration tests to verify that everything works and performance tests to verify that it runs fast. In an ideal world, you’d also have unit tests and integration tests for your Angular code. The good news is JHipster can generate all of this code for you, including integration tests and performance tests. In addition, if you have entities with relationships, it will generate the necessary schema to support them (with foreign keys), and the TypeScript and HTML code to manage them. You can also set up validation to require certain fields, as well as control their length. JHipster supports several methods of code generation. The first uses its entity sub-generator. The entity sub-generator is a command-line tool that prompts you with questions which you answer. JDL-Studio is a browser-based tool for defining your domain model with JHipster Domain Language (JDL). Finally, JHipster-UML is an option for those that like UML. Supported UML editors include Modelio, UML Designer, GenMyModel, and Visual Paradigm. I like the visual nature of JDL-Studio, so I’ll use it for this project. Below is the entity diagram and JDL code needed to generate a simple blog with blogs, entries, and tags. You can click on this URL, or copy/paste the contents of the file below to your hard drive if you’d like to follow along. entity Blog { name String required minlength(3), handle String required minlength(2) } entity Entry { title String required, content TextBlob required, date ZonedDateTime required } entity Tag { name String required minlength(2) } relationship ManyToOne { Blog{user(login)} to User, Entry{blog(name)} to Blog } relationship ManyToMany { Entry{tag(name)} to Tag{entry} } paginate Entry, Tag with infinite-scroll Run the following command (in the blog directory) to import this file. Running this command will generate entities, tests, and a UI. jhipster import-jdl ~/Downloads/jhipster-jdl.jh You’ll be prompted to overwrite src/main/resources/config/liquibase/master.xml. Type a to overwrite this file, as well as others. Start the application with /.mvnw and run yarn start (in another window) to view the UI for the generated entities. Create a couple of blogs for the existing admin and user users, as well as a few blog entries. From these screenshots, you can see that users can see each other’s data, and modify it. Now, check your generated entities into Git. git init git add . git commit -m "Entities generated" Add Business Logic TIP: To configure an IDE with your JHipster project, see Configuring your IDE. Instructions exist for Eclipse, IntelliJ IDEA, Visual Studio Code, and NetBeans. To add more security around blogs and entries, open BlogResource.java and find the getAllBlogs() method. Change the following line: List blogs = blogRepository.findAll(); To: List blogs = blogRepository.findByUserIsCurrentUser(); The findByUserIsCurrentUser() method is generated by JHipster in the BlogRepository class and allows limiting results by the current user. public interface BlogRepository extends JpaRepository { @Query("select blog from Blog blog where blog.user.login = ?#{principal.username}") List findByUserIsCurrentUser(); } After making this change, re-compiling BlogResource should trigger a restart of the application thanks to Spring Boot’s Developer tools. If you navigate to you should only see the blog for the current user. To add this same logic for entries, open EntryResource.java and find the getAllEntries() method. Change the following line: Page page = entryRepository.findAll(pageable); To: Page page = entryRepository.findByBlogUserLoginOrderByDateDesc(SecurityUtils.getCurrentUserLogin(), pageable); Using your IDE, create this method in the EntryRepository class. It should look as follows: Page findByBlogUserLoginOrderByDateDesc(String currentUserLogin, Pageable pageable); Recompile both changed classes and verify that the user user only sees the entries you created for them. After making this changes, commit them to Git. git add . git commit -m "Add business logic" You might notice that this application doesn’t look like a blog and it doesn’t allow HTML in the content field. Make UI Enhancements When doing UI development on a JHipster-generated application, it’s nice to see your changes as soon as you save a file. JHipster 4 uses Browsersync and webpack to power this feature. You enable this previously by running the following command in the blog directory. In this section, you’ll change the following: Change the rendered content field to display HTML Change the list of ent

pass that to a resource server. I mentioned two different flows: getting the authorization and getting the tokens. Those don’t have to happen on the same channel. The front channel is what goes over the browser. The browser redirected the user to the authorization server, the user gave consent. This happens on the user’s browser. Once the user takes that authorization grant and hands that to the application, the client application no longer needs to use the browser to complete the OAuth flow to get the tokens. The tokens are meant to be consumed by the client application so it can access resources on your behalf. We call that the back channel. The back channel is an HTTP call directly from the client application to the resource server to exchange the authorization grant for tokens. These channels are used for different flows depending on what device capabilities you have. For example, a Front Channel Flow where you authorize via user agent might look as follows: Resource Owner starts flow to delegate access to protected resource Client sends authorization request with desired scopes via browser redirect to the Authorize Endpoint on the Authorization Server Authorization Server returns a consent dialog saying “do you allow this application to have access to these scopes?” Of course, you’ll need to authenticate to the application, so if you’re not authenticated to your Resource Server, it’ll ask you to login. If you already have a cached session cookie, you’ll just see the consent dialog box. View the consent dialog, and agree. The authorization grant is passed back to the application via browser redirect. This all happens on the front channel. There’s also a variance in this flow called the implicit flow. We’ll get to that in a minute. This is what it looks like on the wire. Request GET gmail.send &redirect_uri= &response_type=code&client_id=812741506391 &state=af0ifjsldkj This is a GET request with a bunch of query params (not URL-encoded for example purposes). Scopes are from Gmail's API. The redirect_uri is the URL of the client application that the authorization grant should be returned to. This should match the value from the client registration process (at the DMV). You don't want the authorization being bounced back to a foreign application. Response type varies the OAuth flows. Client ID is also from the registration process. State is a security flag, similar to XRSF. To learn more about XRSF, see DZone's "Cross-Site Request Forgery explained". Response HTTP/1.1 302 Found Location: code=MsCeLvIaQm6bTrgtp7&state=af0ifjsldkj The code returned is the authorization grant and state is to ensure it's not forged and it's from the same request. After the Front Channel is done, a Back Channel Flow happens, exchanging the authorization code for an access token. The Client application sends an access token request to the token endpoint on the Authorization Server with confidential client credentials and client id. This process exchanges an Authorization Code Grant for an Access Token and (optionally) a Refresh Token. Client accesses a protected resource with Access Token. Below is how this looks in raw HTTP. Request POST /oauth2/v3/token HTTP/1.1 Host: www.googleapis.com Content-Type: application/x-www-form-urlencoded code=MsCeLvIaQm6bTrgtp7&client_id=812741506391&client_secret={client_secret}&redirect_uri= grant_type=authorization_code The grant_type is the extensibility part of OAuth. It's an authorization code from a precomputed perspective. It opens up the flexibility to have different ways to describe these grants. This is the most common type of OAuth flow. Response { "access_token": "2YotnFZFEjr1zCsicMWpAA", "token_type": "Bearer", "expires_in": 3600, "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA" } The response is JSON. You can be reactive or proactive in using tokens. Proactive is to have a timer in your client. Reactive is to catch an error and attempt to get a new token then. Once you have an access token, you can use the access token in an Authentication header (using the token_type as a prefix) to make protected resource requests. curl -H "Authorization: Bearer 2YotnFZFEjr1zCsicMWpAA" \ So now you have a front channel, a back channel, different endpoints, and different clients. You have to mix and match these for different use cases. This up-levels the complexity of OAuth and it can get confusing. OAuth Flows The very first flow is what we call the Implicit Flow. The reason it’s called the implicit flow is because all the communication is happening through the browser. There is no backend server redeeming the authorization grant for an access token. An SPA is a good example of this flow’s use case. This flow is also called 2 Legged OAuth. Implicit flow is optimized for browser-only public clients. An access token is returned directly from the authorization request (front channel only). It typically does not support refresh tokens. It assumes the Resource Owner and Public Client are on the same device. Since everything happens on the browser, it’s the most vulnerable to security threats. The gold standard is the Authorization Code Flow, aka 3 Legged, that uses both the front channel and the back channel. This is what we’ve been talking about the most in this article. The front channel flow is used by the client application to obtain an authorization code grant. The back channel is used by the client application to exchange the authorization code grant for an access token (and optionally a refresh token). It assumes the Resource Owner and Client Application are on separate devices. It’s the most secure flow because you can authenticate the client to redeem the authorization grant, and tokens are never passed through a user-agent. There’s not just Implicit and Authorization Code flows, there are additional flows you can do with OAuth. Again, OAuth is more of a framework. For server-to-server scenarios, you might want to use a Client Credential Flow. In this scenario, the client application is a confidential client that’s acting on its own, not on behalf of the user. It’s more of a service account type of scenario. All you need is the client’s credentials to do the whole flow. It’s a back channel only flow to obtain an access token using the client’s credentials. It supports shared secrets or assertions as client credentials signed with either symmetric or asymmetric keys. Symmetric-key algorithms are cryptographic algorithms that allow you to decrypt anything, as long as you have the password. This is often found when securing PDFs or .zip files. Public key cryptography, or asymmetric cryptography, is any cryptographic system that uses pairs of keys: public keys and private keys. Public keys can be read by anyone, private keys are sacred to the owner. This allows data to be secure without the need to share a password. There’s also a legacy mode called Resource Owner Password Flow. This is very similar to the direct authentication with username and password scenario and is not recommended. It’s a legacy grant type for native username/password apps such as desktop applications. In this flow, you send the client application a username and password and it returns an access token from the Authorization Server. It typically does not support refresh tokens and it assumes the Resource Owner and Public Client are on the same device. For when you have an API that only wants to speak OAuth, but you have old-school clients to deal with. A more recent addition to OAuth is the Assertion Flow, which is similar to the client credential flow. This was added to open up the idea of federation. This flow allows an Authorization Server to trust authorization grants from third parties such as SAML IdP. The Authorization Server trusts the Identity Provider. The assertion is used to obtain an access token from the token endpoint. This is great for companies that have invested in SAML or SAML-related technologies and allow them to integrate with OAuth. Because SAML assertions are short-lived, there are no refresh tokens in this flow and you have to keep retrieving access tokens every time the assertion expires. Not in the OAuth spec, is a Device Flow. There’s no web browser, just a controller for something like a TV. A user code is returned from an authorization request that must be redeemed by visiting a URL on a device with a browser to authorize. A back channel flow is used by the client application to poll for authorization approval for an access token and optionally a refresh token. Also popular for CLI clients. We’ve covered six different flows using the different actors and token types. They’re necessary because of the capabilities of the clients, how we needed to get consent from the client, who is making consent, and that adds a lot of complexity to OAuth. When people ask if you support OAuth, you have to clarify what they’re asking for. Are they asking if you support all six flows, or just the main ones? There’s a lot of granularity available between all the different flows. Security and the Enterprise There’s a large surface area with OAuth. With Implicit Flow, there’s lots of redirects and lots of room for errors. There’s been a lot of people trying to exploit OAuth between applications and it’s easy to do if you don’t follow recommended Web Security 101 guidelines. For example: Always use CSRF token with the state parameter to ensure flow integrity Always whitelist redirect URIs to ensure proper URI validations Bind the same client to authorization grants and token requests with a client ID For confidential clients, make sure the client secrets aren’t leaked. Don’t put a client secret in your app that’s distributed through an App Store! The biggest complaint about OAuth in general comes from Security people. It’s regarding the Bearer tokens and that they can be passed just like session cookies. You can pass it around and you’re good to go, it’s not cryptographically bound to the user. Using JWTs helps because they can’t be tampered with. However, in the end, a JWT is just a string of characters so they can easily be copied and used in an Authorization header. Enterprise OAuth 2.0 Use Cases OAuth decouples your authorization policy decisions from authentication. It enables the right blend of fine and coarse grained authorization. It can replace traditional Web Access Management (WAM) Policies. It’s also great for restricting and revoking permissions when building apps that can access specific APIs. It ensures only managed and/or compliant devices can access specific APIs. It has deep integration with identity deprovisioning workflows to revoke all tokens from a user or device. Finally, it supports federation with an identity provider. OAuth is not an Authentication Protocol To summarize some of the misconceptions of OAuth 2.0: it’s not backwards compatible with OAuth 1.0. It replaces signatures with HTTPS for all communication. When people talk about OAuth today, they’re talking about OAuth 2.0. Because OAuth is an authorization framework and not a protocol, you may have interoperability issues. There are lots of variances in how teams implement OAuth and you might need custom code to integrate with vendors. OAuth 2.0 is not an authentication protocol. It even says so in its documentation. We’ve been talking about delegated authorization this whole time. It’s not about authenticating the user, and this is key. OAuth 2.0 alone says absolutely nothing about the user. You just have a token to get access to a resource. There’s a huge number of additions that’ve happened to OAuth in the last several years. These add complexity back on top of OAuth to complete a variety of enterprise scenarios. For example, JWTs can be used as interoperable tokens that can be signed and encrypted. Pseudo-Authentication with OAuth 2.0 Login with OAuth was made famous by Facebook Connect and Twitter. In this flow, a client accesses a /me endpoint with an access token. All it says is that the client has access to the resource with a token. People invented this fake endpoint as a way of getting back a user profile with an access token. It’s a non-standard way to get information about the user. There’s nothing in the standards that say everyone has to implement this endpoint. Access tokens are meant to be opaque. They’re meant for the API, they’re not designed to contain user information. What you’re really trying to answer with authentication is who the user is, when did the user authenticate, and how did the user authenticate. You can typically answer these questions with SAML assertions, not with access tokens and authorization grants. That’s why we call this pseudo authentication. Enter OpenID Connect To solve the pseudo authentication problem, the best parts of OAuth 2.0, Facebook Connect, and SAML 2.0 were combined to create OpenID Connect. OpenID Connect (OIDC) extends OAuth 2.0 with a new signed id_token for the client and a UserInfo endpoint to fetch user attributes. Unlike SAML, OIDC provides a standard set of scopes and claims for identities. Examples include: profile, email, address, and phone. OIDC was created to be internet scalable by making things completely dynamic. There’s no longer downloading metadata and federation like SAML requires. There’s built-in registration, discovery, and metadata for dynamic federations. You can type in your email address, then it dynamically discovers your OIDC provider, dynamically downloads the metadata, dynamically know what certs it’s going to use, and allows BYOI (Bring Your Own Identity). It supports high assurance levels and key SAML use cases for enterprises. OIDC was made famous by Google and Microsoft, both big early adopters. Okta has made a big investment in OIDC as well. All that changes in the initial request is it contains standard scopes (like openid and email): Request GET scope=openid email& redirect_uri= response_type=code& client_id=812741506391& state=af0ifjsldkj Response HTTP/1.1 302 Found Location: code=MsCeLvIaQm6bTrgtp7&state=af0ifjsldkj The code returned is the authorization grant and state is to ensure it's not forged and it's from the same request. And the authorization grant for tokens response contains an ID token. Request POST /oauth2/v3/token HTTP/1.1 Host: www.googleapis.com Content-Type: application/x-www-form-urlencoded code=MsCeLvIaQm6bTrgtp7&client_id=812741506391& client_secret={client_secret}& redirect_uri= grant_type=authorization_code Response { "access_token": "2YotnFZFEjr1zCsicMWpAA", "token_type": "Bearer", "expires_in": 3600, "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA", "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ..." } You can see this is layered nicely on top of OAuth to give back an ID token as a structured token. An ID token is a JSON Web Token (JWT). A JWT (aka “jot”) is much smaller than a giant XML-based SAML assertion and can be efficiently passed around between different devices. A JWT has three parts: a header, a body, and a signature. The header says what algorithm was used to sign it, the claims are in the body, and its signed in the signature. An Open ID Connect flow involves the following steps: Discover OIDC metadata Perform OAuth flow to obtain id token and access token Get JWT signature keys and optionally dynamically register the Client application Validate JWT ID token locally based on built-in dates and signature Get additional user attributes as needed with access token OAuth + Okta Okta is best known for its single-sign on services that allow you to seamlessly authenticate to the applications you use on a daily basis. But did you know Okta also has an awesome developer platform? Secure single sign-on often uses SAML as the protocol of choice, but Okta also provides several other options, including a Sign-in Widget, Auth SDK (a JavaScript-based library), Social Login, and an Authentication API for any client. If you’re interested in learning about Okta straight from the source, you should attend Oktane17 in late August. There’s a track dedicated to app development. See Okta’s OIDC/OAuth 2.0 API for specific information on how we support OAuth. SAML is implemented by Okta with its SSO chiclets. If you’re an Okta customer, like me, you likely interact with most apps using something like . When you click on a chiclet, we send a message, we sign the assertion, inside the assertion it says who the user is, and that it came from Okta. Slap on a digital signature on it and you’re good to go. If you’d rather watch a video to learn about OAuth, please see the presentation below from Nate Barbettini, Product Manager at Okta. OAuth 2.0 is an authorization framework for delegated access to APIs. It involves clients that request scopes that Resource Owners authorize/give consent to. Authorization grants are exchanged for access tokens and refresh tokens (depending on flow). There are multiple flows to address varying client and authorization scenarios. JWTs can be used for structured tokens between Authorization Servers and Resource Servers. OAuth has a very large security surface area. Make sure to use a secure toolkit and validate all inputs! OAuth is not an authentication protocol. OpenID Connect extends OAuth 2.0 for authentication scenarios and is often called “SAML with curly-braces”. If you’re looking to dive even deeper into OAuth 2.0, I recommend you check out OAuth.com, take Okta’s Auth SDK for a spin, and try out the OAuth flows for yourself. If you’d like to learn more about OAuth and OIDC, we suggest the following posts: If you’re passionate about OAuth 2.0 and OIDC like we are, give us a follow on Twitter or check out our new security site where we’re publishing in-depth articles on security topics. Page 6 Every good dev knows that time spent setting up the perfect environment and searching out the latest tools is time well spent. Little things make a huge difference — upgrade your IDE plugins, automate a task or two, or look for new tools and libraries that can increase your efficiency. We’ve taken a crack at an updated list of our favorite .NET developer tools, that can simplify your life and amplify your work. 1. JSON.NET Chances are that for any given project these days, you’ll be consuming a RESTful JSON API, producing one, or both. And even if you’re not, you may still be using JSON to serialize data or configuration on disk. Sure, you could get it done using System.Runtime.Serialization.Json, but JSON.NET is faster and has tons of nice features to make your life easier. Built-in LINQ-to-JSON support, plus the ability to query JSON using XPath-like syntax has earned this library the reputation as the de facto JSON implementation for .NET. 2. .NET API Browser (and Reverse Package Search) Here’s a two-for-one for our second tool suggestion: the .NET API Browser and Reverse Package Search. As classes get moved around when their packages are refactored and reshuffled between versions, these two services are invaluable for tracking everything down. Microsoft’s .NET API Browser is a comprehensive reference for all classes and methods in the .NET API. The best part is that it autocompletes while you type, making it easy to locate APIs even when you can’t remember exactly where they exist in the package hierarchy. Similarly, the Reverse Package Search lets you quickly search a vast collection of third party libraries, tagged with their respective supported versions of .NET Framework or .NET Core. It’s a little less curated than Microsoft’s official API browser, but the breadth makes up for it. 3. StyleCop.Analyzers If there’s one thing that can really get a team of developers fussed up, it’s adhering to a consistent code formatting style. The only real solution is to adopt a formatting tool like StyleCop, and then share the configuration throughout the team. StyleCop can detect and fix a broad range of style offences, and can even be set up in some IDEs to work real-time as you type. StyleCop.Analyzers is the latest incarnation of this project. Okta’s very own .NET evangelist Nate is a dedicated contributor. 4. Visual Studio Code Visual Studio Code is Microsoft’s open source text editor and IDE. It’s beautiful and featureful and a joy to use. Not just for .NET, it works well with lots of language ecosystems and runs on Mac, Linux, and Windows. It’s got some similarities to Atom, but comes with IntelliSense, debugger, git integration, and can be extended even further with plugins. 5. Posh-git Been working for days on a major feature branch when a critical issue pops up in production? No worries: just commit to your feature branch, switch to the stable branch to push a fix, and switch back. Git (and modern version control in general) has really enabled this sort of multitasking on a codebase—but it can get confusing quickly. Posh-git (and its equivalent for Subversion, Mercurial, and Perforce) modifies your shell prompt to include the current branch and other status information. It also provides tab-completion for your SCM commands and branch names, which results in a very nice git experience. 6. ReSharper ReSharper is the premier code analysis tool for Visual Studio. It provides analysis, linting, and refactoring assistance. From on-the-fly “quick fixes” to project-wide symbol renaming, this tool is like having your own co-pilot. It costs a few hundred bucks per year, but it should quickly pay for itself in increased productivity, and it comes with a 30-day trial so you can take it for a test drive. Combine it with the StyleCop plugin and get the benefits of code format evaluation as you type. 7. xUnit.net If you’ve invested in good unit test coverage, then you’re undoubtedly pretty committed to a testing framework already. But if you’re starting a new project, or just haven’t got around to writing any tests yet (we won’t judge) then look no further than xUnit.net. It is the successor to NUnit, and builds on the long lineage of similarly-named testing frameworks in other languages such as JUnit, CPPUnit, PHPUnit, and dozens more. However, xUnit has made some opinionated deviations to promote clearer more isolated tests that are easier to maintain, and a focus on data-driven testing using theories. xUnit.net makes writing unit tests more approachable by providing the scaffolding to put them on. It supplies the infrastructure for basic fixturing and test running (including in parallel), and combines nicely with AutoFixture and Moq for writing concise and effective unit tests. Bonus – FluentAssertions On the topic of writing concise and effective unit tests, Fluent Assertions is a library that enables expressive assertions. When a test fails, it’s important to be able to know why without reaching for the debugger. The possibilities for tweaking and refining your development workflow go on and on. While shiny new tools have the potential to dramatically increase productivity, it’s easy to go down the rabbit hole of research and not actually get any work done. So now that you’ve got some new tricks, you can stop reading and go put them to work. Page 7 In the age of the “personalized web experience”, authentication and user management is a given, and it’s easier than ever to tap into third-party authentication providers like Facebook, Twitter, and Google. And it’s not just the wild, wild web that needs it. Businesses need ways to secure their APIs, and identify users logged into their apps. OpenID Connect is a protocol for authenticating users. It is a specification by the OpenID Foundation describing the best way for the authentication “handshake” to happen. It lays out what an Identity Provider needs to provide in order to be considered “OpenID Connect Certified” which makes it easier than ever to consume authentication as a service. Why Not Use OAuth 2.0? First, OAuth 2.0 is NOT an authentication protocol. I know what you’re thinking: “What?!!?” But it’s not. It is an delegated authorization framework, which many modern authentication protocols are built on. Second, while OAuth 2.0 does a great job of providing the necessary information for consumers to make authorization decisions, it says nothing about how that information will be exchanged securely. This has led to every authentication provider having their own way of exchanging the OAuth 2.0 information, which has led to a few well-publicized hacks. OpenID Connect fixes these problems by providing an authentication protocol that describes exactly how the exchange of authorization information happens between a subscriber and their provider. So let’s see how this works. Nothing Up My Sleeve You’ll be using Visual Studio Code and the command line. You can also use Visual Studio, but Visual Studio Code is cross-platform and lightweight, so it won’t matter what operating system you are using. First, get the dotnet command-line program. Once you have the dotnet-cli installed, run the dotnet new mvc --auth None command. This will create a new MVC application without the built-in authentication. You’re going to do that yourself. In just a few seconds you’ll have a simple ASP.NET Core app ready to go. You may need to run a dotnet restore command to install the dependencies for the base application. Add an Identity Provider Now you need an Identity Provider. For this tutorial you’ll use Okta. Once you’ve created an account and logged in, click Admin on the top menu. Then choose the Applications menu item from the admin dashboard. From the Applications page, click the Add Application button. On the Add Application page, click on the Create New Application button. When the New Application Integration window pops up, set up the new application. We’ll choose Web from the Platform dropdown, and OpenID Connect from the Sign On Method. Once you’ve created the application, choose it from the list of applications. In the General Settings tab, click the Edit button and make sure to add to the Redirect URIs. While you are on the General Settings tab, scroll down and copy your Client ID and Client Secret somewhere. You’ll use this later when configuring the middleware in your application. Finally, you’ll need to make sure that you can use the API by adding the localhost URI to your CORS settings. First, choose the Security menu item from the dashboard and the choose API from the drop down menu. Then select the Trusted Origins tab. Click on Add Origin and add as a trusted origin. Make sure that CORS and Redirect are selected. Enter “ASP.NET web app (debug)” as the redirect name (or any other name that makes sense to you). Now you’re ready to set up authentication! Add Authentication When you open your application in Visual Studio or Visual Studio Code, you’ll notice that there’s a .csproj file. You’ll need to make some quick changes to that to get some dependencies you’ll need to make this work. In the ItemGroup section of the csproj document add: This will allow you to use JSON Web Tokens for authorization information, get the tokens from the OpenID Connect provider (Okta in this case) and store them in cookies for session management. You’ll need to run a quick dotnet restore command, but don’t worry, once you save the file, VS Code will give you an option to return to the command line. Now, open the Startup.cs file, and on the first line of the Configure method add: using Microsoft.IdentityModel.Protocols.OpenIdConnect; using System.IdentityModel.Tokens.Jwt; using Microsoft.IdentityModel.Tokens; Then, between the app.UseStaticFiles(); and app.UseMvc(...); add: app.UseCookieAuthentication(new CookieAuthenticationOptions(){ AuthenticationScheme = "Cookies", AutomaticAuthenticate = true }); This tells the application that you want to store your session tokens in cookies. Then, you just need to add the authentication instructions. app.UseOpenIdConnectAuthentication(new OpenIdConnectOptions() { AuthenticationScheme = "oidc", SignInScheme = "Cookies", Authority = "https://{yourOktaDomain}", ResponseType = OpenIdConnectResponseType.Code, ClientId = "{clientId}", ClientSecret = "{clientSecret}", GetClaimsFromUserInfoEndpoint = true, TokenValidationParameters = new TokenValidationParameters { ValidateIssuer = true }, SaveTokens = true }); This is the important part, so let’s go through it line by line: The AuthenticationScheme gives out scheme a name, and will be used to build the default callback url (~/signin-oidc). The SignInScheme is used to set the sign-in middleware. The Authority identifies the authorization endpoint for your Identity Provider. It’s discoverable as part of the OpenID specification, and is located at: https://{yourOktaDomain}/oauth2/default/.well-known/openid-configuration. The ResponseType is also specified in that document under “response_types_supported”. This tells the application you want to start an authorization code flow from from the provider. The ClientId, and ClientSecret are the Client ID and Client Secret you got from the General Settings tab. For production, I would highly suggest storing these in a secure way and referencing them here. They’re in line here for demonstration purposes. Setting GetClaimsFromUserInfoEndpoint = true tells the provider that if you’re successful authenticating, go ahead and make a call to the userinfo_endpoint (specified in the configuration document at the same URL you got the authorization_endpoint and the response_types_supported from). This will get the claims that we’re going to display from Okta once the authentication has completed. The TokenValidationParameters tells the middleware that we want to validate that the issuer is who we expect it to be by getting the signing key from the jwks_uri endpoint in the .well-known/openid_configuration document. Finally, we tell the application to save the token once it comes back from the provider. That’s all there is to it, but how do you know it’s working? Well, you could hook up a login form but there is an easier way! Check Your Work All you really need to do is add an [Authorize] attribute to a controller method. That will simply and quickly show us that it’s going to the login page on the provider, but it won’t show us what the provider is sending back. Let’s create a page that will show that information so you can make sure you’re getting what you think you should be getting from the provider. First, add a using statement to the HomeController.cs for the Authorize attribute. using Microsoft.AspNetCore.Authorization; Then add a “Secure” method to the HomeController.cs controller. [Authorize] public IActionResult Secure() { return View(); } Then create a view to return in ~/Views/Home called Security.cshtml. @{ ViewData["Title"] = "Security"; } Secure @foreach (var claim in User.Claims) { @claim.Type @claim.Value } This will just loop through the claims and output them. When you navigate to you should be redirected to a login page. Once you’re logged in (with a user that is assigned to your app), you should now see a list of the claims on the secure page. If you do, congratulations! You just set up OpenID Connect for authenticating in your ASP.NET Core app! If you have any questions, comments, or suggestions, feel free to reach out to me via email, or hit me up in the comments or via Twitter @leebrandt. Page 8 My favorite thing about Apache Shiro is how easy it makes handling authorization. You can use a role-based access control (RBAC) model of assigning roles to users and then permissions to roles. This makes dealing with the inevitable requirements change simple. Your code does not change, just the permissions associated with the roles. In this post I want to demonstrate just how simple it is, using a Spring Boot application and walking through how I’d handle the following scenario: Your boss (The Supreme Commander) shows up at your desk and tells you the current volunteer (Stormtrooper) registration application needs have different access roles for the different types of employees. Officers can register new “volunteers” Underlings (you and I) only have read access the volunteers Anyone from outside the organization doesn’t have any access to the “volunteers” It should go without saying the Supreme Commander has access to everything Start with a REST Application To get started, grab this Spring Boot example. It’ll get you started with a set of REST endpoints which expose CRUD operations to manage a list of Stormtroopers. You’ll be adding authentication and authorization using Apache Shiro. All of the code is up on GitHub. Using the Apache Shiro Spring Boot starter is all you need, just add the dependency to your pom. (where ${shiro.version} is at least 1.4.0): org.apache.shiro shiro-spring-boot-web-starter ${shiro.version} Jumping into the code we will start with our StormtrooperController, and simply add annotations: @RestController @RequestMapping(path = "/troopers", produces = MediaType.APPLICATION_JSON_UTF8_VALUE) public class StormtrooperController { private final StormtrooperDao trooperDao; @Autowired public StormtrooperController(StormtrooperDao trooperDao) { this.trooperDao = trooperDao; } @GetMapping() @RequiresRoles(logical = Logical.OR, value = {"admin", "officer", "underling"}) public Collection listTroopers() { return trooperDao.listStormtroopers(); } @GetMapping(path = "/{id}") @RequiresRoles(logical = Logical.OR, value = {"admin", "officer", "underling"}) public Stormtrooper getTrooper(@PathVariable("id") String id) throws NotFoundException { Stormtrooper stormtrooper = trooperDao.getStormtrooper(id); if (stormtrooper == null) { throw new NotFoundException(id); } return stormtrooper; } @PostMapping() @RequiresRoles(logical = Logical.OR, value = {"admin", "officer"}) public Stormtrooper createTrooper(@RequestBody Stormtrooper trooper) { return trooperDao.addStormtrooper(trooper); } @PostMapping(path = "/{id}") @RequiresRoles("admin") public Stormtrooper updateTrooper(@PathVariable("id") String id, @RequestBody Stormtrooper updatedTrooper) throws NotFoundException { return trooperDao.updateStormtrooper(id, updatedTrooper); } @DeleteMapping(path = "/{id}") @ResponseStatus(value = HttpStatus.NO_CONTENT) @RequiresRoles("admin") public void deleteTrooper(@PathVariable("id") String id) { trooperDao.deleteStormtrooper(id); } } In the code block above you’re using Shiro’s @RequiresRoles annotation to describe your use-case. You’ll notice the logical OR to allow any of these roles access. This is great, your code is done, it was pretty easy to add, just a single line. You could stop here but, roles are not that flexible, and if you put them directly in your code you’re now tightly coupled to those names/IDs. Stop Using Roles Imagine your application has been deployed and is working fine, the following week your boss stops by your desk and tells you to to make some changes: Officers need to be able to update troopers He feels that the term ‘admin’ is fine for most superiors, but is not suitable to the Dark Lord Fine, you say, easy enough, you can just make a few small changes to the method signatures: @GetMapping() @RequiresRoles(logical = Logical.OR, value = {"emperor", "admin", "emperor", "officer", "underling"}) public Collection listTroopers() @GetMapping(path = "/{id}") @RequiresRoles(logical = Logical.OR, value = {"emperor", "admin", "officer", "underling"}) public Stormtrooper getTrooper(@PathVariable("id") String id) throws NotFoundException @PostMapping() @RequiresRoles(logical = Logical.OR, value = {"emperor", "admin", "officer"}) public Stormtrooper createTrooper(@RequestBody Stormtrooper trooper) @PostMapping(path = "/{id}") @RequiresRoles(logical = Logical.OR, value = {"emperor", "admin", "officer"}) public Stormtrooper updateTrooper(@PathVariable("id") String id, @RequestBody Stormtrooper updatedTrooper) throws NotFoundException @DeleteMapping(path = "/{id}") @ResponseStatus(value = HttpStatus.NO_CONTENT) @RequiresRoles(logical = Logical.OR, value = {"emperor", "admin"}) public void deleteTrooper(@PathVariable("id") String id) After another round of testing the deployment and you’re back in action! Wait, take a step back. Roles are great for simple use cases and making a change like this would work fine, but you know this will be changed again. Instead of changing your code every time the requirements change slightly, let’s decouple the roles and what they represent from your code. Instead, use permissions. Your method signatures will look like this: @GetMapping() @RequiresPermissions("troopers:read") public Collection listTroopers() @GetMapping(path = "/{id}") @RequiresPermissions("troopers:read") public Stormtrooper getTrooper(@PathVariable("id") String id) throws NotFoundException @PostMapping() @RequiresPermissions("troopers:create") public Stormtrooper createTrooper(@RequestBody Stormtrooper trooper) @PostMapping(path = "/{id}") @RequiresPermissions("troopers:update") public Stormtrooper updateTrooper(@PathVariable("id") String id, @RequestBody Stormtrooper updatedTrooper) throws NotFoundException @DeleteMapping(path = "/{id}") @ResponseStatus(value = HttpStatus.NO_CONTENT) @RequiresPermissions("troopers:delete") public void deleteTrooper(@PathVariable("id") String id) By using Shiro’s @RequiresPermissions annotation, this code would work with the original requirements and the new requirements without modification. The only thing that changes is how you map those permissions to roles, and in turn, to users. This could be done externally from your application in a database, or for this example a simple properties file. NOTE: This example uses static usernames and passwords all stored as clear text, this is fine for a blog post, but seriously, manage your passwords correctly! To meet the original requirements, the role-to-permission mapping would look like this: role.admin = troopers:* role.officer = troopers:create, troopers:read role.underling = troopers:read For the updated requirements, you would just change the file slightly to add the new ‘emperor’ role, and grant officers the ‘update’ permission: role.emperor = * role.admin = troopers:* role.officer = troopers:create, troopers:read, troopers:update role.underling = troopers:read If the permission syntax looks a little funny to you, take a look at Apache Shiro’s Wildcard Permission documentation for an in depth explanation. Apache Shiro and Spring We’ve already covered the Maven dependencies and the actual REST controller, but our application will also need a Realm and error handling. If you take a look at the SpringBootApp class you will notice a few things that were NOT in the original example. @Bean public Realm realm() { // uses 'classpath:shiro-users.properties' by default PropertiesRealm realm = new PropertiesRealm(); // Caching isn't needed in this example, but we can still turn it on realm.setCachingEnabled(true); return realm; } @Bean public ShiroFilterChainDefinition shiroFilterChainDefinition() { DefaultShiroFilterChainDefinition chainDefinition = new DefaultShiroFilterChainDefinition(); // use permissive to NOT require authentication, our controller Annotations will decide that chainDefinition.addPathDefinition("/**", "authcBasic[permissive]"); return chainDefinition; } @Bean public CacheManager cacheManager() { // Caching isn't needed in this example, but we will use the MemoryConstrainedCacheManager for this example. return new MemoryConstrainedCacheManager(); } First you have defined a Shiro Realm, a realm is simply a user-store specific DAO. Shiro supports many different types of Realms out of the box (Active Directory, LDAP, Database, file, etc.). Next up you have the ShiroFilterChainDefinition which you’ve configured to allow BASIC authentication but NOT required it by using the ‘permissive’ option. This way your annotations configure everything. Instead of using annotations (or in addition to using them) you could define your permission to URL mappings with Ant-style paths. This example would look something like: chainDefinition.addPathDefinition("/troopers/**", "authcBasic, rest[troopers]"); This would map any resource starting with the path /troopers to require BASIC authentication, and use the ‘rest’ filter which based on the HTTP request method, appends a CRUD action to the permission string. For example an HTTP GET would map to ‘read’ so the full permission string for a ‘GET’ request would be troopers:read (just like you did with your annotations). Exception Handling The last bit of code you have handles exceptions. @ExceptionHandler(UnauthenticatedException.class) @ResponseStatus(HttpStatus.UNAUTHORIZED) public void handleException(UnauthenticatedException e) { log.debug("{} was thrown", e.getClass(), e); } @ExceptionHandler(AuthorizationException.class) @ResponseStatus(HttpStatus.FORBIDDEN) public void handleException(AuthorizationException e) { log.debug("{} was thrown", e.getClass(), e); } @ExceptionHandler(NotFoundException.class) @ResponseStatus(HttpStatus.NOT_FOUND) public @ResponseBody ErrorMessage handleException(NotFoundException e) { String id = e.getMessage(); return new ErrorMessage("Trooper Not Found: "+ id +", why aren't you at your post? "+ id +", do you copy?"); } The first two handle Shiro exceptions and simply set the status to 401 or 403. A 401 for invalid or missing user/passwords, and a 403 for any valid logged in user that does NOT have access to the resource. Lastly, you’ll want to handle any NotFoundException with a 404 and return a JSON serialized ErrorMessage object. Fire it Up! If you put all of this together, or you just grab the code from GitHub, you can start the application using mvn spring-boot:run. Once you have everything running you can start making requests! $ curl HTTP/1.1 401 Content-Length: 0 Date: Thu, 26 Jan 2017 21:12:41 GMT WWW-Authenticate: BASIC realm="application" Don’t forget, you need to authenticate! $ curl --user emperor:secret HTTP/1.1 200 Content-Type: application/json;charset=UTF-8 Date: Thu, 26 Jan 2017 21:14:17 GMT Transfer-Encoding: chunked [{ "id": "FN-0128", "planetOfOrigin": "Naboo", "species": "Twi'lek", "type": "Sand" }, { "id": "FN-1383", "planetOfOrigin": "Hoth", "species": "Human", "type": "Basic" }, { "id": "FN-1692", "planetOfOrigin": "Hoth", "species": "Nikto", "type": "Marine" }, ... A 404 looks would look like this: $ curl --user emperor:secret HTTP/1.1 404 Content-Type: application/json;charset=UTF-8 Date: Thu, 26 Jan 2017 21:15:54 GMT Transfer-Encoding: chunked { "error": "Trooper Not Found: TK-421, why aren't you at your post? TK-421, do you copy?" } Learn More About Apache Shiro This example has shown how easy it is to integrate Apache Shiro into a Spring Boot application, how using permissions allow for greater flexibility over roles, and all it takes is a single Annotation in your controller. At Stormpath we were happy to be able to commit our support to Apache Shiro, and we’ve carried that commitment forward to Okta. Look forward to more Shiro content from our team, including tutorials on using Shiro with Okta and OAuth plus how to add an AngularJS frontend to this volunteer application. Stay tuned, the Empire needs YOU! If you have questions on this example you can send them to Apache Shiro’s user list, me on Twitter, or just leave them in the comments section below! To learn more, check out these posts: Page 9 Progressive Web Apps, aka PWAs, are the best way for developers to make their webapps load faster and more performant. In a nutshell, PWAs are websites that use recent web standards to allow for installation on a user’s computer or device, and deliver an app-like experience to those users. Twitter recently launched mobile.twitter.com as a PWA built with React and Node.js. They’ve had a good experience with PWAs, showing that the technology is finally ready for the masses. In this guide, you’ll learn about the essential ingredients in a PWA, how to install one, why you should build one, and how they stack up against hybrid and native applications. A Deeper Dive – What is a PWA? A PWA is a web application that can be “installed” on your system. It works offline when you don’t have an internet connection, leveraging data cached during your last interactions with the app. If you’re on a desktop, using Chrome, and have the appropriate flags turned on, you will be prompted to install the app when you visit the site. For example, enable the following flags in Chrome by copying and pasting the following URLs into Chrome. chrome://flags/#enable-add-to-shelf chrome://flags/#bypass-app-banner-engagement-checks Click the blue “Relaunch Now” button at the bottom of your browser after enabling these flags. Now if you visit a site like , you’ll see an installation prompt at the top of the page. Click the “Add” button and you’ll see a dialog to name the app, populated with information from the app’s manifest. This adds the application to a “~/Applications/Chrome Apps” directory on a Mac. If you launch the apps, they will run in Chrome rather than having their own icon. On an Android phone, their icon and launch behavior will resemble that of a native application. You can use Chrome’s Developer Tools > Network tab to toggle “Offline” and reload the application. You’ll notice it still loads the data rather than saying it can’t reach the server. Why Should You Build a PWA? You should make your webapp into a PWA because it’ll reduce the time it takes for your app to load and it’ll give your users a better experience. Having it load over HTTPS is a good security practice and adding icons (using a web app manifest) is something you’d do anyway. Having a cache-first service worker strategy will allow your app to work offline (if the user has already loaded data), alleviating one of the biggest issues with webapps. There are a number of other performance recommendations you can implement in your webapp. While the following list is not required for PWAs, many PWAs employ these elements: Implement the PRPL pattern: Push critical resources for the initial URL route. Render initial route. Pre-cache remaining routes. Lazy-load and create remaining routes on demand. Use to tell your browser to load a resource you know you’ll eventually need. This is a W3C Standard specification. Use HTTP/2 and server push to “push” assets to the browser without the user having to ask for them. Use code-splitting and lazy-loading for granular loading of application pages/features. Mariko Kosaka created some drawings to show the difference between HTTP/1 and HTTP/2. I think it illustrates the performance gains provided by HTTP/2 nicely. Note that HTTP/2 requires HTTPS, just like PWAs do. Dynamic Caching and Your PWA If you’re going to build a PWA and leverage service workers, you should become familiar with Chrome Developer Tools’ Application tab. This tab provides the ability to manipulate service workers so they update on reload. If you don’t check this box, developing a PWA will be a frustrating experience. The reason is because everything will be cached in your browser and when you update files in your editor, they won’t be reloaded in your browser. For a great resource on Chrome’s Developer Tools, I recommend Umaar Hunsa’s Dev Tips. Developing PWAs can be painful because it will cause your application to do aggressive caching in your browser. Web developers have been fighting for ages to get the browser to not cache assets, so PWAs kinda go against the grain for web developers. One workaround is to comment out your service worker registration until you package for production. This can be done by sim

when you click the link you would be redirected to login and then redirected back to this same page. On the above screenshot, you see the returned code and original state. That code can now be exchanged for an id_token and an access_token by the middle tier - a Spring Boot application, in this case. This middle tier will validate the state we sent in the authorize request earlier and make a /token request using the Client Secret to mint an access_token and id_token for the user. Implicit Flow The Implicit flow is covered in Section 3.2 of the OIDC spec. Essentially, access and ID tokens are returned directly from the /authorization endpoint. The /token endpoint is not used. This is a suitable approach when working with a client (such as a Single Page Application or mobile application) that you want to interact with the OIDC OP directly. Here's an example of how this flow gets started using Okta: It's almost identical to the authorization code flow, except that the response_type is either id_token, token or id_token+token. Below, we cover exactly what's in these tokens and how it's driven, but remember: an id_token encodes identity information and an access_token (returned if token is specified) is a bearer token used to access resources. Okta also uses JWT for an access_token, which enables additional information to be encoded into it. Here's this flow in the browser: You are redirected back to the redirect_uri originally specified (with the returned tokens and original state): The application can now verify the id_token locally. Use the /introspect endpoint to verify the access_token. It can also use the access_token as a bearer token to hit protected resources, such as the /userinfo endpoint. Hybrid Flow The Hybrid flow is covered in Section 3.3 of the OIDC spec. In this flow, some tokens are returned from the authorization endpoint (/authorize) and others are returned from the token endpoint (/token). This is a suitable approach when you want your end-user application to have immediate access to short-lived tokens – such as the id_token for identity information, and also want to use a backend service to exchange the authorization code for longer-lived tokens using refresh tokens. It's a combination of the authorization code and implicit code flows. You can spot it by looking at the response_type it must contain code and one or both of id_token and token: Here it is in the browser: You are redirected back to the redirect_uri originally specified (with the returned code, tokens and original state): In the next installment, we dig into how to control what's in these tokens, but here's a little taste now: These tokens were produced as a result of hybrid flow with all default scopes enabled. Here's the response from the /userinfo endpoint using the access_token as a bearer token: { "sub": "00u2yulup4eWbOttd1t7", "name": "Okta OIDC Fun", "locale": "en-US", "email": "okta_oidc_fun@okta.com", "preferred_username": "okta_oidc_fun@okta.com", "given_name": "Okta OIDC", "family_name": "Fun", "zoneinfo": "America/Los_Angeles", "updated_at": 1499922371, "email_verified": true } Other OIDC Flows There are two other flows not covered in this post: Client Credentials Flow and Resource Owner Password Credentials. These are both defined in the OAuth 2.0 spec and, as such, are supported by OIDC. Here, we're focusing on flows that require an external authentication provider, such as Okta or Google, and not the alternative methods that these flows support. What information is encoded in the id_token, the access_token and what information is returned when hitting the protected /userinfo endpoint are a function of the flow type and the scopes requested. In the next post, we dig deeper into this. Token Teardowns In this installment, we looked at OIDC in action. You can experiment with OIDC at: You can easily create your own instance of the OIDC tool if you have an Okta tenant. Check mine out at: You can explore the code or just click the friendly purple button to deploy your own instance. In the final installment, we dig into the various types of tokens and how to validate them. The whole series is live now. Part 1 is here. Part 3 is here. If you enjoyed this, you might also want to check out our new security site where we're publishing lots of in-depth infosec articles. Page 12 In the previous two installments of this OpenID Connect (OIDC) series, we dug deep into the OIDC flow types and saw OIDC in action using a playground found at: . In this third and final installment, we'll look at what's encoded into the various types of tokens and how to control what gets put in them. JWTs, have the benefit of being able to carry information in them. With this information available to your app you can easily enforce token expiration and reduce the number of API calls. Additionally, since they're cryptographically signed, you can verify that they have not been tampered with. The source code that backs the site can be found at: . There are two primary sources for information relating to identity as dictated by the OIDC spec. One source is the information encoded into the id_token JWT. Another is the response from the /userinfo endpoint, accessible using an access_token as a bearer token. At Okta, we've chosen to make our access tokens JWTs as well, which provides a third source of information. (You'll see this in many OIDC implementations.) There are a lot of combinations of query parameters in the /authorization request that determine what information will be encoded into an id_token. The two query parameters that impact what will ultimately be found in returned tokens and the /userinfo endpoint are response_type and scope. OIDC Response Types For the moment, we'll set aside scope and focus on response_type. In the following examples, we use only the scopes, openid (required) and email. We'll also work with the implicit flow, since that gives us back tokens immediately. Given this request: Notice that response_type=token will yield us an access_token. A particular format is not required in the OIDC spec for access tokens, but at Okta we use JWTs. Looking inside the returned token, we see: { "active": true, "scope": "openid email", "username": "okta_oidc_fun@okta.com", "exp": 1501531801, "iat": 1501528201, "sub": "okta_oidc_fun@okta.com", "aud": "test", "iss": " , "jti": "AT.upPJqU-Ism6Fwt5Fpl8AhNAdoUeuMsEgJ_VxJ3WJ1hk", "token_type": "Bearer", "client_id": "0oa2yrbf35Vcbom491t7", "uid": "00u2yulup4eWbOttd1t7" } This is mainly resource information, including an expiration (exp) and a user id (uid). If we want to get identity information for the user, we must hit the /userinfo endpoint using the access_token as a bearer token. Here's what that looks like using HTTPie: http Authorization:"Bearer eyJhbGciOiJSUzI1NiIsImtpZCI6Ik93bFNJS3p3Mmt1Wk8zSmpnMW5Dc2RNelJhOEV1elY5emgyREl6X3RVRUkifQ..." HTTP/1.1 200 OK ... { "sub": "00u2yulup4eWbOttd1t7", "email": "okta_oidc_fun@okta.com", "email_verified": true } We get back the sub, email and email_verified claims. This is because of the default scope=openid+email from the original request. We'll look at some more detailed responses in the scopes section. Let's try another request: This time, I'm asking for an ID token by using response_type=id_token. The response is a JWT (as required by the OIDC spec) with this information encoded into it: { "sub": "00u2yulup4eWbOttd1t7", "email": "okta_oidc_fun@okta.com", "ver": 1, "iss": " , "aud": "0oa2yrbf35Vcbom491t7", "iat": 1501528456, "exp": 1501532056, "jti": "ID.4Mmzy2kj5_B8nGZ_PT4dt8-fzu1tA2W3C5dbEF-N6Us", "amr": ["pwd"], "idp": "00o1zyyqo9bpRehCw1t7", "nonce": "c96fa468-ca1b-46f0-8974-546f23f9ee6f", "email_verified": true, "auth_time": 1501528157 } Notice that we have the sub and emailclaims encoded directly in the JWT. In this type of implicit flow, we have no bearer token to use against the /userinfo endpoint, so the identity information is baked right into the JWT. Finally, let's look at the last type of implicit flow: Here, we are requesting both an id_token and an access_token in the response. Our access_token has the same claims as before. The id_token has the following: { "sub": "00u2yulup4eWbOttd1t7", "email": "okta_oidc_fun@okta.com", "ver": 1, "iss": " , "aud": "0oa2yrbf35Vcbom491t7", "iat": 1501528536, "exp": 1501532136, "jti": "ID.fyybPizTmYLoQR20vlR7mpo8WTxB7JwkxplMQom-Kf8", "amr": ["pwd"], "idp": "00o1zyyqo9bpRehCw1t7", "nonce": "c96fa468-ca1b-46f0-8974-546f23f9ee6f", "auth_time": 1501528157, "at_hash": "T7ij7o69gBtjo6bAJvaVBQ" } Notice that there's less information in the id_token this time (in this case, there's no email_verified claim). Because we also requested the access_token, it's expected that we will get the rest of the available identity information (based on scope) from the /userinfo endpoint. In this case, it yields the same information as before when we only requested the access_token OIDC Scopes Combining all the available scopes with all the possible response types yields a large set of information to present: 48 combinations, to be exact. First, I'll enumerate what each scope yields and then we'll look at a few real world examples combining request_type and scope. The first thing to note is that the different scopes have an impact on the information encoded in an id_token and returned from the /userinfo endpoint. Here's a table of scopes and resultant claims. More information can be found in Section 5.4 of the OIDC Spec scope resultant claims openid (required for all OIDC flows) profile name, family_name, given_name, middle_name, nickname, preferred_username profile (cont'd) profile, picture, website, gender, birthdate, zoneinfo, locale, updated_at email email, email_verified address address phone phone_number, phone_number_verified Let's try each of our implicit flows with all the possible (default) scope types. The only difference in the resultant access_token compared to before is that all the scopes are encoded into the scp array claim. This time, when I use the access_token to hit the /userinfo endpoint, I get back more information: http Authorization:"Bearer eyJhbGciOiJSUzI1NiIsImtpZCI6Ik93bFNJS3p3Mmt1Wk8zSmpnMW5Dc2RNelJhOEV1elY5emgyREl6X3RVRUkifQ..." HTTP/1.1 200 OK ... { "sub": "00u2yulup4eWbOttd1t7", "name": "Okta OIDC Fun", "locale": "en-US", "email": "okta_oidc_fun@okta.com", "preferred_username": "okta_oidc_fun@okta.com", "given_name": "Okta OIDC", "family_name": "Fun", "zoneinfo": "America/Los_Angeles", "updated_at": 1499922371, "email_verified": true } Note: While it's not the complete list of claims defined from profile scope, it's all the claims for which my user in Okta has a value. Let's try just the id_token implicit flow (still with all the default scopes): Here's what's encoded into the id_token I get back: { "sub": "00u2yulup4eWbOttd1t7", "name": "Okta OIDC Fun", "locale": "en-US", "email": "okta_oidc_fun@okta.com", "ver": 1, "iss": " , "aud": "0oa2yrbf35Vcbom491t7", "iat": 1501532222, "exp": 1501535822, "jti": "ID.Zx8EclaZmhSckGHOCRzOci2OaduksmERymi9-ad7ML4", "amr": ["pwd"], "idp": "00o1zyyqo9bpRehCw1t7", "nonce": "c96fa468-ca1b-46f0-8974-546f23f9ee6f", "preferred_username": "okta_oidc_fun@okta.com", "given_name": "Okta OIDC", "family_name": "Fun", "zoneinfo": "America/Los_Angeles", "updated_at": 1499922371, "email_verified": true, "auth_time": 1501528157 } All the (available) identity information is encoded right into the token, since I don't have a bearer token to hit the /userinfo endpoint with. Finally, let's try the last variant of the Implicit Flow: response_type=id_token+token: In this case, we have some of the claims encoded into the id_token: { "sub": "00u2yulup4eWbOttd1t7", "name": "Okta OIDC Fun", "email": "okta_oidc_fun@okta.com", "ver": 1, "iss": " , "aud": "0oa2yrbf35Vcbom491t7", "iat": 1501532304, "exp": 1501535904, "jti": "ID.1C2NQext2hM0iJy55cLc_Ryc45urVYC1wJ0S-KebkpI", "amr": ["pwd"], "idp": "00o1zyyqo9bpRehCw1t7", "nonce": "c96fa468-ca1b-46f0-8974-546f23f9ee6f", "preferred_username": "okta_oidc_fun@okta.com", "auth_time": 1501528157, "at_hash": "GB5O9CpSSOUSfVZ9CRekRg", "c_hash": "mRNStYQm-QU4rwcfv88VKA" } If we use the resultant access_token to hit the /userinfo endpoint, in this case, we get back: http Authorization:"Bearer eyJhbGciOiJSUzI1NiIsImtpZCI6Ik93bFNJS3p3Mmt1Wk8zSmpnMW5Dc2RNelJhOEV1elY5emgyREl6X3RVRUkifQ..." HTTP/1.1 200 OK ... { "sub": "00u2yulup4eWbOttd1t7", "name": "Okta OIDC Fun", "locale": "en-US", "email": "okta_oidc_fun@okta.com", "preferred_username": "okta_oidc_fun@okta.com", "given_name": "Okta OIDC", "family_name": "Fun", "zoneinfo": "America/Los_Angeles", "updated_at": 1499922371, "email_verified": true } This rounds out all the identity information that was requested in the scopes. Custom Scopes and Claims The OIDC spec accommodate custom scopes and claims. The ability to include custom claims in a token (which is cryptographically verifiable) is an important capability for identity providers. Okta's implementation provides support for this. The screenshot below shows my Authorization Server's Claims tab: Clicking the "Add Claim" button brings up a dialog: In the above screenshot, the custom claim is defined using Okta's Expression Language. Unique to Okta, the expression language is a flexible way to describe rules for building a property to include (or not) in custom claims. Using the implicit flow with response_type=id_token and scope=openid+profile, we now get back an id_token with these claims encoded in it: { "sub": "00u2yulup4eWbOttd1t7", "ver": 1, "iss": " , "aud": "0oa2yrbf35Vcbom491t7", "iat": 1501533536, "exp": 1501537136, "jti": "ID.TsKlBQfGmiJcl2X3EuhzyyLfmzqi0OCd66rJ3Onk7FI", "amr": ["pwd"], "idp": "00o1zyyqo9bpRehCw1t7", "nonce": "c96fa468-ca1b-46f0-8974-546f23f9ee6f", "auth_time": 1501528157, "at_hash": "hEjyn3mbKjuWanuSAF-z4Q", "full_name": "Okta OIDC Fun" } Notice the full_name claim present in the id_token. Verifying Tokens Access tokens can be verified by hitting the /introspect endpoint. For an active token, you get a response like this: http --auth : -f POST \ \ token=eyJhbGciOiJSUzI1NiIsImtpZCI6Ik93bFNJS3p3Mmt1Wk8zSmpnMW5Dc2RNelJhOEV1elY5emgyREl6X3RVRUkifQ... HTTP/1.1 200 OK ... { "active": true, "aud": " , "client_id": "xdgqP32nYN148gn3gJsW", "exp": 1498517509, "fullName": "Micah Silverman", "iat": 1498513909, "iss": " , "jti": "AT.JdXQPAuh-JTqhspCL8nLe2WgbfjcK_-jmlp7zwaYttE", "scope": "openid profile", "sub": "micah+okta@afitnerd.com", "token_type": "Bearer", "uid": "00u9vme99nxudvxZA0h7", "username": "micah+okta@afitnerd.com" } Since it requires the OIDC client ID and secret, this operation would typically be done in an application server where it's safe to have those credentials. You would not want something like an end-user web or mobile application to have access to the OIDC client secret. If the token parameter is invalid or expired, the /introspect endpoint returns this: http --auth : -f POST \ \ token=bogus HTTP/1.1 200 OK ... { "active": false } ID tokens can be verified using the JWK endpoint. JWK is a JSON data structure that represents a crypto key. The JWK endpoint is exposed from the OIDC "well known" endpoint used for API discovery. This returns a lot of information. Here's an excerpt: http HTTP/1.1 200 OK ... { "authorization_endpoint": " , ... "introspection_endpoint": " , ... "issuer": " , "jwks_uri": " , ... "userinfo_endpoint": " } Some of the endpoints, such as /userinfo and /authorize, should look familiar by now. The one we're interested in is the /keys endpoint shown in jwks_uri. http HTTP/1.1 200 OK ... { "keys": [{ "alg": "RS256", "e": "AQAB", "kid": "cbkhWG0YmFsGiNO1LEkWSEszDCTNfwvJPpXxuVf_kX0", "kty": "RSA", "n": "g2XQgdyc5P6F4K26ioKiUzrdgfy90eBgIbcrKkspKZmzRJ3CIssv69f1ClJvT784J-...", "use": "sig" }] } Notice the kid claim. It matches the kid claim in the header from our id_token: { "typ": "JWT", "alg": "RS256", "kid": "cbkhWG0YmFsGiNO1LEkWSEszDCTNfwvJPpXxuVf_kX0" } We can also see that the algorithm used is RS256. Using the public key found in the n claim along with a security library, we can confirm that the ID token has not been tampered with. All of this can be done safely on an end-user SPA, mobile app, etc. Here's a Java example that uses the claims from the jwks_uri above to verify an id_token: java -jar target/jwk-token-verifier-0.0.1-SNAPSHOT-spring-boot.jar \ eyJhbGciOiJSUzI1NiIsImtpZCI6Ik93bFNJS3p3Mmt1Wk8zSmpnMW5Dc2RNel... \ g2XQgdyc5P6F4K26ioKiUzrdgfy90eBgIbcrKkspKZmzRJ3CIssv69f1ClJvT784J-... \ AQAB Verified Access Token { "header" : { "alg" : "RS256", "kid" : "cbkhWG0YmFsGiNO1LEkWSEszDCTNfwvJPpXxuVf_kX0" }, "body" : { "ver" : 1, "jti" : "AT.LT9cRL_Kzd3T8Izw_ONZxHJ5xGBPD0m13iiEIDK_Nbw", "iss" : " , "aud" : "test", "iat" : 1501533536, "exp" : 1501537136, "cid" : "0oa2yrbf35Vcbom491t7", "uid" : "00u2yulup4eWbOttd1t7", "scp" : ["openid"], "sub" : "okta_oidc_fun@okta.com" }, "signature" : "ZV_9tYxt4v4bp9WEEDu038b7v_OHsbMZw13daR1s5_tI56oayBgJlnqf-..." } If any part of the id_token JWT had been tampered with, you would see this instead: io.jsonwebtoken.SignatureException: JWT signature does not match locally computed signature. JWT validity cannot be asserted and should not be trusted. Verifying JWT's using the /introspect endpoint and using JWKs is a powerful component of OIDC. It allows for a high degree of confidence that the token has not been tampered in any way. And, because of that, information contained within – such as expiration – can be safely enforced. How I Learned to Love OpenID Connect When OIDC was first released and early implementers, such as Google, adopted it, I thought: "I just got used to OAuth 2.0. Why do I have to learn a new thing that rides on top of it?" It took some time, but here is what I consider to be the important takeaways: OIDC formalizes a number of things left open in OAuth 2.0. Things like: specific token formats (id_token) and specific scopes and claims. There's explicit support for Authentication and Authorization. OAuth 2.0 was always presented purely as an authorization framework, but people would get confused with certain flows that allowed for authentication. There's a clear separation between identity (id_token and /userinfo) and access to resources (access_token). The different flows provide clean use case implementations for mobile apps, SPAs, and traditional web apps. It's inherently flexible. It's easy to provide custom scopes and claims and to dictate what information should be encoded into tokens beyond the default specification. All the code used in this series can be found on github. You can use the OIDC sample app to exercise the various flows and scopes discussed throughout these posts. It's at: . The entire final OIDC spec can be found here. And you can learn more about OAuth 2.0 at oauth.com. The whole series is live now. Part 1 is here. Part 2 is here. If you're looking for other security-focused articles, you may want to check out our new security site, where we're publishing lots of other in-depth security pieces. Page 13 You've built a microservices architecture with Spring Boot and Spring Cloud. You're happy with the results, and you like how it adds resiliency to your application. You're also pleased with how it scales and how different teams can deploy microservices independently. But what about security? Are you using Spring Security to lock everything down? Are your microservices locked down too, or are they just behind the firewall? This tutorial shows you how you can use Spring Security, Okta, and a few Java libraries to secure your microservices architecture. Not only that, but I'll show you how to secure everything, so even your backend services communicate securely. You'll learn how to use JWTs and Juiser to read an X-Forwarded-User header and turn it into a Spring Security User. This tutorial builds off Build a Microservices Architecture for Microbrews with Spring Boot. A simple microservices architecture with Spring Boot and Spring Cloud looks as follows. It uses Stormpath's Spring Boot Starter (it's been modified to work with Okta while we work on building up Okta's Java support) and Juiser, a library created by Les Hazlewood. Juiser is independent and open source, and is not tied to a particular identity provider. Once you've completed this tutorial, you'll have Spring Security locking things down, and Okta providing authentication and JWT validation. In this tutorial, you'll build a microservices architecture with Spring Boot and related projects. To add security with Okta, you'll have to create two applications in your developer console. The first will be a "Native" application that supports the Stormpath Java SDK, and OAuth grant types authorization code, refresh token, and resource owner password. This type of application is typically reserved for native mobile applications, but it also includes the Stormpath Java SDK. This is because the Stormpath SDK was retrofitted to work with Okta, and not built specifically for the Okta API, per se. For the Angular client, you'll need a second "SPA" application. To begin, you'll need to clone the aforementioned article's completed project. git clone -b v1.0 Create an Okta Developer account, and create a "Native" application that works with the Stormpath Spring Boot Starter. Here's an abbreviated list of steps: In the developer console, navigate to Applications > Add Application Click Native and Next On the next page, enter the following values and click Done Application Name: My Test App Login redirect URIs: After your application has been created, you still have a few settings you need to change: On the General tab, click Edit on the General Settings panel Select Refresh Token and Resource Owner Password and click Save Click Edit on the Client Credentials panel Select Use Client Authentication and click Save Copy and save the Client ID for your application At this point, your application's settings should look as follows: You'll also need to create an API token: On the top menu, click on API > Tokens Click Create Token On the modal, give your new token a name, for example: Microservices, and click Create Token Important: You will need to remember this token value, so copy/paste it somewhere safe. After completing these steps, you should have the information you need to set the following environment variables. export STORMPATH_CLIENT_BASEURL=https://{yourOktaDomain} export OKTA_APPLICATION_ID={clientId} export OKTA_API_TOKEN={apiToken} export OKTA_AUTHORIZATIONSERVER_ID=default Add Stormpath's Zuul Support to the Edge Service NOTE: I'm using Stormpath's Java SDK in this example. It has been updated to work with Okta's API. If you'd like to see how to do this same tutorial using Spring Security and its OAuth support, please Secure a Spring Microservices Architecture with OAuth 2.0. The edge-service application handles the routing to the backend beer-catalog-service, so it's the best place to start securing things. Add the Stormpath BOM (Bill Of Materials) in the dependencyManagement section of edge-service/pom.xml. com.stormpath.sdk stormpath-bom 2.0.4-okta pom import Then add a dependency for Stormpath's Zuul integration. com.stormpath.spring stormpath-zuul-spring-cloud-starter Add the following properties and values to edge-server/src/main/resources/application.properties. server.use-forward-headers=true zuul.routes.beer-catalog-service.path=/beers zuul.routes.beer-catalog-service.url= zuul.routes.home.path=/home zuul.routes.home.url= stormpath.web.cors.allowed.originUris= stormpath.zuul.account.header.jwt.key.resource=classpath:rsatest.priv.pem # This is just one example of a key ID - anything that the origin server can make sense of to lookup # the corresponding public key is fine. Here we use the public key file name. stormpath.zuul.account.header.jwt.key.id=rsatest.pub.pem Copy the rsatest.* files from the Stormpath Zuul example project, or create new ones using the following command: openssl genrsa -out rsatest.priv.pem 2048 Generate the private key's corresponding rsatest.pub.pem public key with: openssl rsa -in rsatest.priv.pem -pubout > rsatest.pub.pem After copying (or generating), both rsatest.priv.pem and rsatest.pub.pem files should be in edge-service/src/main/resources. Add Juiser to the Beer Catalog Service Juiser is a small Java library that automates token authentication during an HTTP request. In this example, Juiser reads the X-Forwarded-User header and creates a Spring Security User for you. For Juiser to read the JWT sent by Stormpath's Zuul support, you need to copy the public key (rsatest.pub.pem) from edge-service/src/main/resources to beer-catalog-service/src/main/resources. Then add the following dependencies to the Beer Catalog Service's pom.xml. ... 1.56 1.0.0 org.juiser juiser-spring-boot-starter ${juiser.version} org.bouncycastle bcpkix-jdk15on ${bouncycastle.version} runtime org.springframework.boot spring-boot-starter-security org.springframework.security spring-security-config org.springframework.security spring-security-web org.springframework.boot spring-boot-starter-thymeleaf ... Create a HomeController in src/main/java/com/example/beercatalogservice/HomeController.java to render the user's information so you can verify authentication is working. package com.example.beercatalogservice; import org.juiser.model.User; import org.springframework.stereotype.Controller; import org.springframework.ui.Model; import org.springframework.web.bind.annotation.GetMapping; @Controller public class HomeController { private final User user; public HomeController(User user) { this.user = user; } @GetMapping("/home") public String howdy(Model model) { model.addAttribute("user", user); return "home"; } } NOTE: There is an issue with Juiser 1.0.0 that it won't initialize if you don't have at least one @Controller in your project. Create a home.html template in beer-catalog-service/src/main/resources/templates/home.html and populate it with the following code. th { text-align: left; } td { white-space: nowrap; } td:first-child { font-family: "Courier", monospace; font-size: 0.9em; color: #343434; } Hello Joe! Login User Properties Name Value anonymous authenticated href id name givenName middleName familyName nickname username profile picture website email emailVerified gender birthdate zoneInfo phoneNumber phoneNumberVerified createdAt updatedAt Add the following properties to beer-catalog-service/src/main/resources/application.properties to configure Juiser. server.use-forward-headers=true juiser.header.jwt.key.resource=classpath:rsatest.pub.pem Create a SecurityConfig.java class in the same package as HomeController. This class configures Spring Security so it secures all endpoints. package com.example.beercatalogservice; import org.springframework.context.annotation.Configuration; import org.springframework.security.config.annotation.web.builders.HttpSecurity; import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter; @Configuration public class SecurityConfig extends WebSecurityConfigurerAdapter { @Override protected void configure(HttpSecurity http) throws Exception { http.authorizeRequests().anyRequest().fullyAuthenticated(); } } Add RequestInterceptor for Feign The @FeignClient used to talk to beer-catalog-service is not aware of the X-Forwarded-User header. To make it aware, create a ForwardedAccountRequestInterceptor class in the same directory as EdgeServiceApplication. package com.example.edgeservice; import com.stormpath.sdk.servlet.http.Resolver; import com.stormpath.zuul.account.ForwardedAccountHeaderFilter; import feign.RequestInterceptor; import feign.RequestTemplate; import org.slf4j.Logger; import org.slf4j.LoggerFactory; import org.springframework.web.context.request.RequestContextHolder; import org.springframework.web.context.request.ServletRequestAttributes; import javax.servlet.http.HttpServletRequest; import javax.servlet.http.HttpServletResponse; public class ForwardedAccountRequestInterceptor implements RequestInterceptor { private static final Logger LOGGER = LoggerFactory.getLogger(ForwardedAccountRequestInterceptor.class); private final Resolver valueResolver; public ForwardedAccountRequestInterceptor(Resolver accountStringResolver) { this.valueResolver = accountStringResolver; } @Override public void apply(RequestTemplate template) { if (template.headers().containsKey(ForwardedAccountHeaderFilter.DEFAULT_HEADER_NAME)) { LOGGER.warn("The X-Forwarded-User has been already set"); } else { LOGGER.debug("Constructing Header {} for Account", ForwardedAccountHeaderFilter.DEFAULT_HEADER_NAME); HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolder.getRequestAttributes()).getRequest(); HttpServletResponse response = ((ServletRequestAttributes) RequestContextHolder.getRequestAttributes()).getResponse(); template.header(ForwardedAccountHeaderFilter.DEFAULT_HEADER_NAME, valueResolver.get(request, response)); } } } Register it as a @Bean in EdgeServiceApplication. import org.springframework.context.annotation.Bean; import feign.RequestInterceptor; import org.springframework.beans.factory.annotation.Qualifier; import com.stormpath.sdk.servlet.http.Resolver; ... public class EdgeServiceApplication { public static void main(String[] args) { SpringApplication.run(EdgeServiceApplication.class, args); } @Bean public RequestInterceptor forwardedAccountRequestInterceptor(@Qualifier("stormpathForwardedAccountHeaderValueResolver") Resolver accountStringResolver) { return new ForwardedAccountRequestInterceptor(accountStringResolver); } } While you're modifying EdgeServiceApplication, change the HystrixCommand in BeerController to make Hystrix execute on the calling thread (so it's aware of the security context). import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty; ... @HystrixCommand(fallbackMethod = "fallback", commandProperties = { @HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE") }) NOTE: There is an issue with the Stormpath Zuul Starter where it doesn't work with Spring Cloud Edgeware.SR1. Change edge-service/pom.xml to have Dalston.RELEASE to workaround this issue. Verify Secure Communication Verify communication between the edge-service and beer-catalog-service works by starting all the Spring Boot applications. First, start eureka-service: cd eureka-service ./mvnw spring-boot:run In a new terminal window, start beer-catalog-service: cd beer-catalog-service ./mvnw spring-boot:run In another terminal window, start edge-service: cd edge-service ./mvnw spring-boot:run Open your browser and navigate to . You should see a login page, prompting for your credentials. This page is served up from the stormpath-zuul-spring-cloud-starter using Thymeleaf. Spring Boot auto-activates Thymeleaf when it finds it in the classpath. After logging in, you should see a page displaying your user's information. Click the Logout button to delete the cookies in your browser and end your session. To use Okta's Sign-In Widget, you'll need to create an additional app in Okta, this time for a SPA (Single-Page Application). Create an OIDC app in Okta by going to the Developer Console and navigating to Applications and click on the Add Application button. Select SPA and click Next. On the next page, specify as a Base URI, Login redirect URI, and Logout redirect URI. Click Done and you should see settings like the following. Install Okta's Sign-In Widget to make it possible to communicate with the secured server. cd client npm install npm install @okta/okta-signin-widget --save Add the widget's CSS to src/styles.css: @import '~@okta/okta-signin-widget/dist/css/okta-sign-in.min.css'; @import '~@okta/okta-signin-widget/dist/css/okta-theme.css'; Create client/src/app/shared/okta/okta.service.ts and use it to configure the widget to talk to your Okta instance. import { Injectable } from '@angular/core'; import * as OktaSignIn from '@okta/okta-signin-widget'; @Injectable() export class OktaService { widget; constructor() { this.widget = new OktaSignIn({ baseUrl: 'https://{yourOktaDomain}', clientId: '{clientId}', authParams: { issuer: 'default', responseType: ['id_token', 'token'], scopes: ['openid', 'email', 'profile'] } }); } getWidget() { return this.widget; } getIdToken() { return this.widget.tokenManager.get('idToken'); } getAccessToken() { return this.widget.tokenManager.get('accessToken'); } } Make sure to replace {yourOktaDomain} and {clientId} in the above code. Add OktaService as a provider to client/src/app/app.module.ts. import { OktaService } from './shared/okta/okta.service'; @NgModule({ ... providers: [OktaService], bootstrap: [AppComponent] }) export class AppModule { } Modify client/src/app/shared/beer/beer.service.ts to read the access token and set it in an Authorization header when it exists. import { Injectable } from '@angular/core'; import { HttpClient, HttpHeaders } from '@angular/common/http'; import { Observable } from 'rxjs/Observable'; import { OktaService } from '../okta/okta.service'; @Injectable() export class BeerService { constructor(private http: HttpClient, private oktaService: OktaService) { } getAll(): Observable { let headers: HttpHeaders = new HttpHeaders(); if (this.oktaService.getAccessToken()) { const accessToken = this.oktaService.getAccessToken(); // headers is immutable, so re-assign headers = headers.append('Authorization', accessToken.tokenType + ' ' + accessToken.accessToken); } return this.http.get(' , {headers: headers}); } } Modify app.component.html to add a placeholder for the widget and a section to show the user's name and a logout button. Welcome to {{title}}! Welcome {{user?.name}}! Logout You'll notice the user variable in the HTML. To resolve this, you need to change your src/app/app.component.ts so it renders the Sign-In Widget. Angular's ChangeDetectorRef is used to notify Angular when things have changed and rendering needs to process changed variables. import { ChangeDetectorRef, Component, OnInit } from '@angular/core'; import { OktaService } from './shared/okta/okta.service'; @Component({ selector: 'app-root', templateUrl: './app.component.html', styleUrls: ['./app.component.css'] }) export class AppComponent implements OnInit { title = 'app'; user; signIn; constructor(private oktaService: OktaService, private changeDetectorRef: ChangeDetectorRef) { this.signIn = oktaService.getWidget(); } showLogin() { this.signIn.renderEl({el: '#okta-signin-container'}, (response) => { if (response.status === 'SUCCESS') { response.forEach(token => { if (token.idToken) { this.signIn.tokenManager.add('idToken', token); this.user = this.getUser(token); } if (token.accessToken) { this.signIn.tokenManager.add('accessToken', token); } }); this.signIn.remove(); this.changeDetectorRef.detectChanges(); }); } getUser(token) { return { name: token.claims.name, email: token.claims.email, username: token.claims.preferred_username }; } ngOnInit() { this.signIn.session.get((response) => { if (response.status !== 'INACTIVE') { const token = this.oktaService.getIdToken(); this.user = this.getUser(token); this.changeDetectorRef.detectChanges(); } else { this.showLogin(); } }); } logout() { this.signIn.signOut(() => { this.user = undefined; this.changeDetectorRef.detectChanges(); this.showLogin(); }); } } In order for the BeerListComponent (at src/app/beer-list/beer-list.component.ts) to detect that you've logged in, you need to use add a constructor dependency on ChangeDetectorRef and invoke its detectChanges() method when you set the giphyUrl property on each beer. import { ChangeDetectorRef, Component, OnInit } from '@angular/core'; import { BeerService, GiphyService } from '../shared'; @Component({ selector: 'app-beer-list', templateUrl: './beer-list.component.html', styleUrls: ['./beer-list.component.css'], providers: [BeerService, GiphyService] }) export class BeerListComponent implements OnInit { beers: Array; constructor(private beerService: BeerService, private giphyService: GiphyService, private changeDetectorRef: ChangeDetectorRef) { } ngOnInit() { this.beerService.getAll().subscribe(data => { this.beers = data; for (const beer of this.beers) { this.giphyService.get(beer.name).subscribe(url => { beer.giphyUrl = url; this.changeDetectorRef.detectChanges(); }); } }, error => console.log(error)) } } Verify Authentication Works Start the client with npm start, navigate to , and you should see a login form like the following. NOTE: If it logs you in automatically, this is likely because you have cookies for still in your browser. Clear your cookies, or try an incognito window. If you want to adjust the style of the form, so it isn't right up against the top toolbar, add the following to client/src/styles.css. #okta-signin-container { margin-top: 25px; } You should be able to log in, see a welcome message, as well as a logout button. Learn More This article showed you how to use Spring Security, Okta, and a few Java libraries to secure a microservices architecture. With JWTs, Zuul, Spring Security, and Juiser, you can ensure your backend services communicate securely. The source code for this tutorial is available on GitHub, in the "okta" branch. git clone git checkout okta Learn more about Okta and its APIs at developer.okta.com. If you have questions about this tutorial, please hit me up on Twitter @mraible or post a question to Stack Overflow with an "okta" tag. Update: To learn how to lock down this application with Spring Security and OAuth, see Secure a Spring Microservices Architecture with Spring Security and OAuth 2.0. Page 14 Need to decouple your web service and client? You're probably using REST endpoints, and if you're a Java shop you've probably tried out JAX-RS, Spring REST, or both. But is one better than the other? In this post I'll go over the differences between the two using basically the same code for an apples to apples comparison. In future posts I'll show you how easy it is to secure these REST endpoints using Apache Shiro and Okta. Lay Down the Foundation – Model and DAO To keep things focused, I'll leave the Maven dependencies out of this post. You can browse the full source on GitHub, the pom files should be self explanatory: one for JAX-RS, another for Spring. First up, we need to get the common bits out of the way. A simple model and DAO (Data Access Object) will be used in all of the examples to register and manage Stormtrooper objects. public class Stormtrooper { private String id; private String planetOfOrigin; private String species; private String type; public Stormtrooper() { // empty to allow for bean access } public Stormtrooper(String id, String planetOfOrigin, String species, String type) { this.id = id; this.planetOfOrigin = planetOfOrigin; this.species = species; this.type = type; } ... // bean accessor methods The Stormtrooper object contains an id and a few other attributes: planetOfOrigin, species, and type. The DAO interface is just as simple, with the basic CRUD methods and an additional list method: public interface StormtrooperDao { Stormtrooper getStormtrooper(String id); Stormtrooper addStormtrooper(Stormtrooper stormtrooper); Stormtrooper updateStormtrooper(String id, Stormtrooper stormtrooper); boolean deleteStormtrooper(String id); Collection listStormtroopers(); } The actual implementation of the StormtrooperDao is not important for these examples, If you are interested, you can take a look at code for DefaultStormtrooperDao, which generates 50 random Stormtroopers. Try Spring Now that we have the common bits out of the way, we can get into the meat of our Spring example. A basic Spring Boot app doesn't get much easier than this: @SpringBootApplication public class SpringBootApp { @Bean protected StormtrooperDao stormtrooperDao() { return new DefaultStormtrooperDao(); } public static void main(String[] args) { SpringApplication.run(SpringBootApp.class, args); } } There are a few things to point out: The @SpringBootApplication annotation sets up Spring's auto configuration and classpath scanning of components @Bean binds an instance of DefaultStormtrooperDao to the StormtrooperDao interface The main method starts the application uses the SpringApplication.run() helper method to bootstrap the application Spring Controller Next up, we have the implementation of our REST endpoint, or in the Spring world a Controller. We will use this class to map our DAO to incoming HTTP requests. @RestController @RequestMapping("/troopers") public class StormtrooperController { private final StormtrooperDao trooperDao; @Autowired public StormtrooperController(StormtrooperDao trooperDao) { this.trooperDao = trooperDao; } @GetMapping public Collection listTroopers() { return trooperDao.listStormtroopers(); } @GetMapping("/{id}") public Stormtrooper getTrooper(@PathVariable("id") String id) throws NotFoundException { Stormtrooper stormtrooper = trooperDao.getStormtrooper(id); if (stormtrooper == null) { throw new NotFoundException(); } return stormtrooper; } @PostMapping public Stormtrooper createTrooper(@RequestBody Stormtrooper trooper) { return trooperDao.addStormtrooper(trooper); } @PostMapping("/{id}") public Stormtrooper updateTrooper(@PathVariable("id") String id, @RequestBody Stormtrooper updatedTrooper) throws NotFoundException { return trooperDao.updateStormtrooper(id, updatedTrooper); } @DeleteMapping("/{id}") @ResponseStatus(value = HttpStatus.NO_CONTENT) public void deleteTrooper(@PathVariable("id") String id) { trooperDao.deleteStormtrooper(id); } } Let's break this down: @Controller @RequestMapping("/troopers") public class StormtrooperController { The @RestController is a convenience annotation for both @Controller and @ResponseBody which marks this class as a web component discovered during classpath scanning. An @RequestMapping annotation at the the class

and helps both us and our users. Regional community presence For those developers that happen to live in the same town as our DevRel Team, there are even more benefits. We sponsor meetups that use our favorite technologies, and we're interested in sponsoring more! We also have a company-wide initiative that allows us to buy developers a strong cup of coffee or an ice cold beer. In exchange, we hope to hear about your pain points with APIs like ours, and how we can make things easier for you. Our DevRel Team is traveling to many big cities in the US to give tech talks, hang out, and generally get to know YOU. We do this so we can solicit feedback from other developers, build relationships, and try to positively impact developers lives all over the place. If you're interested in having us speak at your user group outside of the US, we can make that happen too! However, at this time, we require you to cover our travel expenses. We sponsor meetups because we recognize that they're an important source of knowledge for many developers around the world. They also offer a sense of community, networking, and social interaction that can't be found online. Influencer programs We're just getting started with our influencer program. We hope to engage with influencers in our respective communities and help them out by doing the little things that make them feel welcome. For example, giving them rides from the airport, taking them out to dinner, and hosting community meetups. If you're an influencer in your community and have ideas, let us know! We do this because we know how important influencers are and how traveling solo can be lonely. We want to make your hard work and traveling experience more pleasant! Support and engagement in open source tooling Supporting open source is something we did well at Stormpath. We helped support the Java JWT project, created a plugin for Chrome to inspect JWTs, and actively engaged with the open source security projects in the Java community (e.g. Apache Shiro and Spring Security) to make them better. At Okta, we plan to do much of the same. We're working to make better, and we plan to do the same for OpenID Connect. We'll continue to invest in the Java JWT project, and we're working closely with the Spring Security team to make sure their OpenID Connect support is world-class. If you're involved in an open source project that helps developers use OAuth, OIDC, and other authentication standards, let us know how we can help! We have the budget to help with hosting costs, promotional materials, or whatever your project might need. Internal advocacy We're doing our best to educate Okta employees how to work with developers too. We've already done several presentations on what DevRel is, and we're making moves to make our documentation better, and our blog friendlier. We know that Developer DNA cannot just be a part of the DevRel team. All of Okta needs Developer DNA. We've been telling people they can help by sharing feedback with us, having a beer with an engineer, build for good developer experience, and by giving back to the community. DevEx, DevRel, and ROI The DevRel Team focuses on making developers' lives easier, but it's not all fun and games. Our fearless leader, Alex Salazar, is in charge of making sure our efforts are right on the business side of things. Alex is the former CEO of Stormpath, so he's hyper aware of revenue growth, cost reduction, and trying to balance predictability and risk. He knows that our go-to-market strategy needs to be around developer experience (DevEx) and developer relations (DevRel). It's going to be tough to market to developers if our product is crap. We're focused as a team on cost efficiency and repeatability, to increase our ROI. We know what types of DevRel are the most effective (blogging, how to videos). We know what DevEx investments will have the greatest impact (ease of use, excellent documentation). We'll do more of what's working, and stop doing the things that aren't. And more than anything, we're tracking the metrics that matter. This way, we know how many developers are using our API, with which SDKs, and how long it takes them to activate their account after signing up. Here's to the Future! Being a part of the DevRel Team at Okta has been a great experience for me. It's fun to be a part of something that's just beginning, with a team of outstanding individuals. If you're a developer using Okta, and you have feedback to share, please contact us at developers@okta.com. DevRel at Okta is a fun ride, and we're just getting started. If you'd like to talk to me about DevRel at Okta, or in your organization, please hit me up on Twitter @mraible or directly via email to matt.raible@okta.com. Page 17 With Okta and OpenID Connect (OIDC) you can easily integrate authentication into an Ionic application, and never have to build it yourself again. OIDC allows you to authenticate directly against the Okta API, and this article shows you how to do just that in an Ionic application. I'll demo how to log in with OIDC redirect, using Okta's Auth SDK as well as how to use OAuth with Cordova's in-app browser; user registration is omitted as the feature is still under active development. Why Ionic? Ionic is an open source mobile SDK for developing native and progressive web applications. It leverages Angular and Apache Cordova to allow you to build mobile apps with HTML, CSS, and JavaScript. Apache Cordova embeds the HTML code inside a native WebView on the device, using a foreign function interface to access the native resources of it. You might've heard of PhoneGap - this is Adobe's commercial version of Cordova. Cordova and PhoneGap allow you to target multiple platforms (e.g. Android and iOS) with one codebase. Not only that, but the apps look like native apps and perform just as well. If you need to tap into native features that aren't available in web technologies, there are a number of native plugins. Ionic Native is a curated set of these plugins. I first started using Ionic in late 2013. The project I was working on was developing a native application, but wanted to build several screens of the application with HTML so web developers could author them. I wrote about my experience in March 2014. I enjoyed working with it and found that porting an existing app to use it was more about modifying HTML and tweaking CSS. Ionic 2 was released in January, making it possible to develop Ionic applications with Angular. Ionic 3 was released in April, allowing development with Angular 4. NOTE: "Angular" is the common name for Angular 2+. AngularJS is the common name for the 1.x versions. The reason for #ItsJustAngular is Angular 4 was released in March 2017. See Branding Guidelines for Angular and AngularJS for more information. This article will show you how to build a simple Ionic application and add user authentication to it. Most applications require authentication, so they know who the user is. Once an app knows who you are, it can save your data and better personalization features. Get Started with Ionic To set up your environment to develop with Ionic, complete the following steps: Install Node.js Install Ionic and Cordova using npm: npm install -g cordova ionic Create an Ionic Application From a terminal window, create a new application using the following command: You will be prompted to select a starter project. For this tutorial, choose the tabs starter project. When prompted to integrate your app with Cordova, answer yes. When prompted to install the free Ionic Pro SDK and connect your app, answer no. Project creation may take a minute or two to complete, depending on your internet connection speed. Run the commands below to start your Ionic application. cd ionic-auth ionic serve This command will open your default browser on . You can use Chrome's device toolbar to see what the application will look like on an iPhone 6. One slick feature of Ionic's serve command is it shows compilation errors in the browser, rather than in the (sometimes hidden) developer console. For example, if you give an invalid type to the rootPage variable in app.component.ts, you'll see an error like the one below. Add User Authentication Ionic Cloud offers a free Auth service. It allows authentication with an email and password, as well as social providers like Facebook, Google, and Twitter. It provides several classes you can use to build authentication in its @ionic/cloud-angular dependency. It even has support for custom authentication, but it "requires your own server to handle authentication" and will be decommissioned on January 31, 2018. While there aren't many current tutorials on using this service, there are a few from last year. You might notice that both tutorials require quite a bit of code. Also, there doesn't seem to be a lot of documentation on how you can verify user credentials from the Auth service in a backend service. Create an OpenID Connect App in Okta OpenID Connect (OIDC) builds on top of the OAuth 2.0 protocol. It allows clients to verify the identity of the user and obtain their basic profile information. To integrate Okta's Identity Platform for user authentication, you'll first need to: Register and create an OIDC application Log in to your Okta account and navigate to Applications > Add Application Select SPA and click Next Give your application a name (e.g. "Ionic OIDC") Change the Base URI and Login redirect URI to and click Done. You should see settings like the following: Create a Login Page Generate a login page for authentication by running the following command: In the generated src/pages/login/login.html, add a form with username and password fields. Login {{error}} Login You can leverage a couple of open source libraries to perform the actual authentication. The first one is Manfred Steyer's angular-oauth2-oidc. This library allows you to interact with identity and access tokens easily. The second is the Okta Auth SDK. OAuth is not an authentication protocol, but OIDC is. Why is it necessary to add Okta's authentication library then? Because OIDC authentication works via redirect (when using in a SPA) and I'd rather perform authentication without redirecting to Okta. Install angular-oauth2-oidc and the Okta Auth SDK using npm. npm install angular-oauth2-oidc @okta/okta-auth-js --save In src/pages/login/login.ts, add the basic structure of the LoginPage class and a constructor that configures your OIDC settings with the OAuthService from angular-oauth2-oidc. You will need to replace {clientId} with the Client ID from your Okta OIDC settings and {yourOktaDomain} with your account's correct URI. import { Component, ViewChild } from '@angular/core'; import { IonicPage, NavController } from 'ionic-angular'; import { JwksValidationHandler, OAuthService } from 'angular-oauth2-oidc'; @IonicPage() @Component({ selector: 'page-login', templateUrl: 'login.html' }) export class LoginPage { @ViewChild('email') email: any; private username: string; private password: string; private error: string; constructor(private navCtrl: NavController, private oauthService: OAuthService) { oauthService.redirectUri = window.location.origin; oauthService.clientId = '{clientId}'; oauthService.scope = 'openid profile email'; oauthService.issuer = 'https://{yourOktaDomain}/oauth2/default'; oauthService.tokenValidationHandler = new JwksValidationHandler(); // Load Discovery Document and then try to login the user this.oauthService.loadDiscoveryDocument().then(() => { this.oauthService.tryLogin(); }); } ionViewDidLoad(): void { setTimeout(() => { this.email.setFocus(); }, 500); } } Modify src/app/app.component.ts to check to see if the user is logged in. If they're not, set the LoginPage as the rootPage. import { Component } from '@angular/core'; import { Platform } from 'ionic-angular'; import { StatusBar } from '@ionic-native/status-bar'; import { SplashScreen } from '@ionic-native/splash-screen'; import { TabsPage } from '../pages/tabs/tabs'; import { OAuthService } from 'angular-oauth2-oidc'; import { LoginPage } from '../pages/login/login'; @Component({ templateUrl: 'app.html' }) export class MyApp { rootPage: any = TabsPage; constructor(platform: Platform, statusBar: StatusBar, splashScreen: SplashScreen, oauthService: OAuthService) { if (oauthService.hasValidIdToken()) { this.rootPage = TabsPage; } else { this.rootPage = LoginPage; } platform.ready().then(() => { statusBar.styleDefault(); splashScreen.hide(); }); } } Update src/app/app.module.ts to add OAuthModule, HttpClientModule, and LoginPageModule as imports. import { LoginPageModule } from '../pages/login/login.module'; import { OAuthModule } from 'angular-oauth2-oidc'; @NgModule({ ... imports: [BrowserModule, LoginPageModule, OAuthModule.forRoot(), IonicModule.forRoot(MyApp)], ... }) Run ionic serve to make sure the LoginPage is displayed when the app first loads. You'll see the following error when the app tries to load: No provider for HttpClient! This error happens because OAuthService has a dependency on Angular's Http, but it hasn't been imported into your project. Add HttpModule as an import in src/app/app.module.ts. import { HttpClientModule } from '@angular/common/http'; @NgModule({ ... imports: [... HttpClientModule, ...], ... }) Now the login screen should load. You can use Chrome's Device Toolbar to see what it'll look like on an iPhone 6. Add a login() method in src/app/pages/login/login.ts that uses the Okta Auth SDK to 1) login and 2) exchange the session token for an identity and access token. An ID token is similar to an identity card, in standard JWT format, signed by the OpenID Provider. Access tokens are part of the OAuth specification. An access token can be a JWT. They are used to access protected resources, often by setting them as an Authentication header when making a request. import * as OktaAuth from '@okta/okta-auth-js'; import { TabsPage } from '../tabs/tabs'; ... login(): void { this.oauthService.createAndSaveNonce().then(nonce => { const authClient = new OktaAuth({ clientId: this.oauthService.clientId, redirectUri: this.oauthService.redirectUri, url: 'https://{yourOktaDomain}', issuer: 'default' }); return authClient.signIn({ username: this.username, password: this.password }).then((response) => { if (response.status === 'SUCCESS') { return authClient.token.getWithoutPrompt({ nonce: nonce, responseType: ['id_token', 'token'], sessionToken: response.sessionToken, scopes: this.oauthService.scope.split(' ') }) .then((tokens) => { const idToken = tokens[0].idToken; const accessToken = tokens[1].accessToken; const keyValuePair = `#id_token=${encodeURIComponent(idToken)}&access_token=${encodeURIComponent(accessToken)}`; this.oauthService.tryLogin({ customHashFragment: keyValuePair, disableOAuth2StateCheck: true }); this.navCtrl.push(TabsPage); }); } else { throw new Error('We cannot handle the ' + response.status + ' status'); } }).fail((error) => { console.error(error); this.error = error.message; }); }); } You want an identity token so you can have more information about the user. You want an access token so you can use it to access protected APIs that require a Bearer token. For example, in Adding Authentication to Your Angular PWA, there's a BeerService that sends an access token when it makes an API request. import { Injectable } from '@angular/core'; import { Http, Response, Headers, RequestOptions } from '@angular/http'; import 'rxjs/add/operator/map'; import { Observable } from 'rxjs'; import { OAuthService } from 'angular-oauth2-oidc'; @Injectable() export class BeerService { constructor(private http: Http, private oauthService: OAuthService) { } getAll(): Observable { const headers: Headers = new Headers(); headers.append('Authorization', this.oauthService.authorizationHeader()); let options = new RequestOptions({ headers: headers }); return this.http.get(' , options) .map((response: Response) => response.json()); } } You can (optionally), pretty up the login screen by adding a logo above the form. Download this image, copy it to src/assets/imgs/okta.png, and add the following above the tag in login.html. You might want to edit the image so it's only 300 pixels wide. This reduces its size to 12 KB instead of 110 KB. When you sign in, there's not much proof on the UI. Add a "Logout" button in the top right corner of the home screen. Replace the in src/pages/home/home.html with the HTML below. Home Logout In src/pages/home/home.ts, add a logout() method, as well as methods to get a name and claims from the identity token. Claims in an ID token are bits of information about the issuer, the user, intended audience, expiration date, and issue date. You can see the standard claims in the OIDC spec. import { Component } from '@angular/core'; import { App } from 'ionic-angular'; import { LoginPage } from '../login/login'; import { OAuthService } from 'angular-oauth2-oidc'; @Component({ selector: 'page-home', templateUrl: 'home.html' }) export class HomePage { constructor(public app: App, public oauthService: OAuthService) { } logout() { this.oauthService.logOut(true); this.app.getRootNavs()[0].setRoot(LoginPage); } get givenName() { const claims: any = this.oauthService.getIdentityClaims(); if (!claims) { return null; } return claims.name; } get claims() { return this.oauthService.getIdentityClaims(); } } To display this information on the home tab, add the following HTML just after the second paragraph in src/app/home/home.html. You are logged in as: {{ givenName }} **Claims from Identity Token JWT:** {{claims | json}} Update src/app/home/home.scss to add some CSS to make the raw JSON look a bit better. page-home { .claims { pre { color: green; } } pre { border: 1px solid silver; background: #eee; padding: 10px; } } Now you should see your name and claims information displayed when you log in. You should also be able to log out and see the login screen with its logo. Deploy to a Mobile Device It's pretty cool that you're able to develop mobile apps with Ionic in your browser. However, it's nice to see the fruits of your labor and validate how awesome your app looks on a phone. It does look and behave like a native app! To see how your application will look on different devices you can run ionic serve --lab. The --lab flag opens a page in your browser that lets you see how your app will display on various devices. The LoginPage tries to auto-focus onto the email field when it loads. To auto-activate the keyboard you'll need to tell Cordova it's OK to display the keyboard without user interaction. You can do this by adding the following to config.xml in the root directory. iOS To emulate or deploy to an iOS device, you'll need a Mac and a fresh installation of Xcode. If you'd like to build iOS apps on Windows, Ionic offers an Ionic Package service. Make sure to open Xcode to complete the installation. Then run ionic cordova emulate ios to open your app in Simulator. You may encounter an error like the following: ** BUILD SUCCEEDED ** Error: Cannot read property 'replace' of undefined [ERROR] An error occurred while running cordova emulate ios (exit code 1). This is a known issue and can be solved by running the following commands: cd platforms/ios/cordova && npm install ios-sim TIP: The biggest problem I found when running the app in Simulator was that it was difficult to get the keyboard to pop up. To workaround this, I used Hardware > Keyboard > Toggle Software Keyboard when I needed to type text in a field. If you enter your credentials on the login screen you'll notice nothing happens. Open Safari and navigate to Develop > Simulator > MyApp / Login, and you'll see that eventually, an error shows up in the console. If you don't see a Develop menu, review the steps in this article to enable it. If you use the Network tab, you can see that only one network request is made (to /authn), which is different than the two requests (to /authn and /authorize) that are made when running in a browser. I believe this doesn't work when the app is packaged with Cordova because it's making a request to the server with an embedded iframe that then posts back to the current window using postMessage. It seems that Ionic/Cordova doesn't support this flow (yet). To work around this issue, you can talk directly to Okta's OAuth service using an in-app browser that's provided by Cordova. Nic Raboy shows how to do this with Facebook in Using An OAuth 2.0 Service Within An Ionic 2 Mobile App. Install the Cordova In-App Browser plugin using the following command: ionic cordova plugin add cordova-plugin-inappbrowser Open src/app/pages/login/login.html and wrap the with a that only shows this login form when running in a browser. Add a new that is displayed when running in an emulator or on a device. ... Login with Okta Open src/pages/login/login.ts and add a reference to window just below the imports. declare const window: any; Then add the methods below to facilitate logging in with OAuth. redirectLogin() { this.oktaLogin().then(success => { const idToken = success.id_token; const accessToken = success.access_token; const keyValuePair = `#id_token=${encodeURIComponent(idToken)}&access_token=${encodeURIComponent(accessToken)}`; this.oauthService.tryLogin({ customHashFragment: keyValuePair, disableOAuth2StateCheck: true }); this.navCtrl.push(TabsPage); }, (error) => { this.error = error; }); } oktaLogin(): Promise { return this.oauthService.createAndSaveNonce().then(nonce => { let state: string = Math.floor(Math.random() * 1000000000).toString(); if (window.crypto) { const array = new Uint32Array(1); window.crypto.getRandomValues(array); state = array.join().toString(); } return new Promise((resolve, reject) => { const oauthUrl = this.buildOAuthUrl(state, nonce); const browser = window.cordova.InAppBrowser.open(oauthUrl, '_blank', 'location=no,clearsessioncache=yes,clearcache=yes'); browser.addEventListener('loadstart', (event) => { if ((event.url).indexOf(' ') === 0) { browser.removeEventListener('exit', () => {}); browser.close(); const responseParameters = ((event.url).split('#')[1]).split('&'); const parsedResponse = {}; for (let i = 0; i < responseParameters.length; i++) { parsedResponse[responseParameters[i].split('=')[0]] = responseParameters[i].split('=')[1]; } const defaultError = 'Problem authenticating with Okta'; if (parsedResponse['state'] !== state) { reject(defaultError); } else if (parsedResponse['access_token'] !== undefined && parsedResponse['access_token'] !== null) { resolve(parsedResponse); } else { reject(defaultError); } } }); browser.addEventListener('exit', function (event) { reject('The Okta sign in flow was canceled'); }); }); }); } buildOAuthUrl(state, nonce): string { return this.oauthService.issuer + '/v1/authorize?' + 'client_id=' + this.oauthService.clientId + '&' + 'redirect_uri=' + this.oauthService.redirectUri + '&' + 'response_type=id_token%20token&' + 'scope=' + encodeURI(this.oauthService.scope) + '&' + 'state=' + state + '&nonce=' + nonce; } Change the redirectUri that's set in the constructor to hard-code . If you skip this step window.location.origin will result in a file:// origin being sent when the app is running on a device. By making it a known URL, we can look for it with the in-app browser on the "loadstart" event. constructor(private navCtrl: NavController, private oauthService: OAuthService) { oauthService.redirectUri = ' '; ... } You'll have to re-deploy your app to your phone after making these changes. ionic cordova emulate ios Now you should be able to log in by clicking on the "Login with Okta" button and entering valid credentials. The nice thing about using this technique is the Okta login screen has Remember Me and Forgot Password support, so you don't need to code those yourself. To deploy the app to an iPhone, start by plugging one into your computer. Then run the following commands to build the app and run it on your device. This command will likely fail if you haven't previously set up code signing for your application. Signing for "MyApp" requires a development team. Select a development team in the project editor. Code signing is required for product type 'Application' in SDK 'iOS 10.3' Open your project in Xcode using the command below. open platforms/ios/MyApp.xcodeproj Ionic's deployment documentation provides instructions to solve this issue. Select your phone as the target in Xcode and click the play button to run your app. The first time you do this, Xcode may spin for a while with a "Processing symbol files" message at the top. Once you've configured your phone, computer, and Apple ID, you should be able to open the app and log in. Below is how it looks on my iPhone. Android To emulate or deploy to an Android device, you'll first need to install Android Studio. As part of the install, it will show you where it installed the Android SDK. Set this path as an ANDROID_HOME environment variable. On a Mac, it should be ~/Library/Android/sdk/. If you've just installed Android Studio, make sure to open it to complete the installation. To deploy to the Android emulator, run ionic cordova emulate android. This command will install Android support and display an error if you don't have any AVD (Android Virtual Device) images. (node:9300) UnhandledPromiseRejectionWarning: CordovaError: No emulator images (avds) found. 1. Download desired System Image by running: /Users/mraible/Library/Android/sdk/tools/android sdk 2. Create an AVD by running: /Users/mraible/Library/Android/sdk/tools/android avd HINT: For a faster emulator, use an Intel System Image and install the HAXM device driver To create a new AVD, open Android Studio and navigate to Tools > Android > AVD Manager. Create a new Virtual Device and click Play. I chose a Pixel 2. After performing these steps, you should be able to run ionic cordova emulate android and see your app running in the AVD. NOTE: If you get an application error that says "The connection to the server was unsuccessful. (file:///android/www/index.html)", add the following line to config.xml. This line sets the default timeout to 60 seconds (default is 20). Thanks to the Stack Overflow community for this solution. PWAs with Ionic Ionic ships with support for creating progressive web apps (PWAs). This means you can deploy your Ionic app as a web app (rather than a mobile app) and make it run offline in browsers that support service workers. You can see how to enable service workers and make your app into a PWA by reading the PWAs section of how to develop a mobile app with Ionic and Spring Boot. A PWA is a web application that can be "installed" on your system. It works offline when you don't have an internet connection, leveraging data cached during your last interactions with the app. Adding PWA features can make your apps load a lot faster, creating happy users. To learn more about PWAs, see The Ultimate Guide to Progressive Web Applications. Ionic has invested heavily in supporting PWAs. You can read more about why in What Progressive Web Apps can do for you. Learn More I hope you've enjoyed this tour of Ionic, Angular, and Okta. I like how Ionic takes your web development skills up a notch and allows you to create mobile applications that look and behave natively and perform swiftly. You can see the complete source code for this project on GitHub. Please contact me on Twitter @mraible or on Okta's Developer Forums if you have any questions. To learn more about Ionic, Angular, or Okta, please see the following resources: Page 18 Developers use APIs to for everything! You build APIs for your own apps to consume, or as a part of a microservices architecture. Bottom line, you're building and using APIs to make your life easier. The ongoing effort to simplify development and work more efficiently, sometimes this also means looking for new libraries or processes (or more often less process). For many teams managing authentication and access control for their apps and APIs is more work than it's worth, or simply not an efficient use of time, so we want to share a few tips that will save you time and code, along with making your applications more secure and easier to maintain. For a bit of context: Okta at its core, is a Java-based REST+JSON API, built on the Spring Framework. We store user credentials and data on behalf of other companies, so for us security is paramount. Thus, my first requirement for these tips is that they help manage access to your Java API securely. These tips should be universal to any type of Java application. They will help you move faster, write less code, and at the same time be more secure: a trifecta! 1. Don't roll your own security Seriously, just don't, it's hard. Almost everyone knows to avoid implementing their own cryptography. The rest of your security stack is no different, and the risk/reward just isn't worth it. There's a high chance you'll make some sort of mistake. Since 1999 there have been 89373 CVEs (Common Vulnerabilities and Exposures). And that's just what's been made public, many of those by very smart people. You may think that dealing with a simple use case like validating a user's password is trivial, all you're doing is just comparing a couple strings after all. You would be wrong. You need to validate the password's hash, audit the attempt, mitigate against dictionary attacks, and that's just the tip of the iceberg. Your best bet is to use an existing library or a framework like Apache Shiro or Spring Security and let the framework handle the complexities! 2. Use TLS, always! It's 2017, everything should be HTTPS now, even the sites on your company's intranet. Let's encrypt makes this free and easy, which means you can stop using insecure self-signed keys too! You can even set up a local Tomcat or Nginx instance with a certificate. Making your application require TLS (HTTPS/SSL) is usually a one liner, so everybody should be doing it! For Apache Shiro, it is just property: And Spring Security, a single method call when configuring an HttpSecurity: http.requiresChannel() .anyRequest().requiresSecure(); Or just use a few properties with Spring Boot: server.port=8443 server.ssl.key-store=classpath:keystore.jks server.ssl.key-store-password=secret server.ssl.key-password=another-secret 3. Build your Java web service with Spring Boot Spring Boot is an opinionated view of the Spring platform which makes it dead simple to write twelve-factor apps in very few lines. If you're still building WAR files you owe it to yourself to check this out. You can create complicated, application wide functions like setting up an OAuth resource server by using a single annotation (@EnableResourceServer) or change the server's port with a single property: If Spring is not your bag, take a look at Dropwizard for an opinionated JAX-RS stack. 4. Use monitoring and metrics to watch your back It's pretty difficult to pinpoint errors without any data. Spring Boot makes gathering metrics easy with Actuator, just add a single dependency to your application. org.springframework.boot spring-boot-starter-actuator Then browse to /health or /metrics to view health checks or application metrics respectively. Dropwizard does the same thing with /healthcheck and /metrics. Here's an output from a Spring Boot application's /metrics endpoint, out of the box: { "classes": 7704, "classes.loaded": 7704, "classes.unloaded": 0, "counter.status.200.metrics": 1, "gauge.response.metrics": 99.0, "gc.ps_marksweep.count": 2, "gc.ps_marksweep.time": 272, "gc.ps_scavenge.count": 8, "gc.ps_scavenge.time": 136, "heap": 3728384, "heap.committed": 470016, "heap.init": 262144, "heap.used": 207793, "httpsessions.active": 0, "httpsessions.max": -1, "instance.uptime": 25020, "mem": 529086, "mem.free": 262222, "nonheap": 0, "nonheap.committed": 60608, "nonheap.init": 2496, "nonheap.used": 59067, "processors": 8, "systemload.average": 5.56103515625, "threads": 24, "threads.daemon": 22, "threads.peak": 28, "threads.totalStarted": 32, "uptime": 37182 } 5. Protect your sensitive bits People treat API keys insecurely, it's a fact of life. Keys get emailed around or checked into source control. Maybe this is because they seem more opaque than a password, I don't know, but they're just as sensitive, if not more so. If you need to store your API keys in a file, make sure there is limited access to that file. For example, we recommend storing our Okta yaml file in private directory ~/.okta/okta.yaml and setting the file permissions to allow only the owner to read: $ chmod u=r,go-rwx ~/.okta/okta.yaml If you are creating API keys for users of your applications, plan to warn them. SSH ignores files in your ~/.ssh directory if the permissions are not set correctly. GitHub does a great job of warning users by marking items in the UI with 'Danger Zone' marking. Bonus: Write less code, give Okta a try! Java has a bit of a reputation (and rightly so) for being verbose. All of the examples above show you how to write less code and when possible take advantage of existing libraries so you can focus on the code that will drive your business. Shameless plug time: You can also write less code by integrating Okta for fully featured user management. Just connect your apps, choose an IdP (or use ours), add users, configure rules, customize your login page, and then gain insights from our built-in reports. Want to see Okta in action? Check out these tutorials: And, as always, if you have any questions or comments you can hit me up on Twitter @briandemers, or follow our whole team @oktadev. Page 19 Creating a consistent environment for development, testing, staging, and production is one of the big benefits of using containers. Not only do containers make the entire environment portable, they remove environment-specific problems, like, "Why does it work in test, but not in production?" Usually, it's a package or framework that's installed on the test machine that is not on the production server. Containers carry all those dependencies with them, minimizing the possibility for those problems. To help create a consistent container, you need an image that is configured in code that can be versioned and distributed. That's where the Dockerfile comes in. A Dockerfile (without an extension) is simply a text file with some keywords and rules that Docker uses to create an image. That image is then used to create a container, or multiple containers that all have the same set up. In this tutorial, you'll build a Dockerfile that you'll use to create an image for a basic web application. In the previous article in this series, I told you that images are like blueprints for creating containers. Well really, they are containers. Containers frozen in time that you can use to "stamp out a copy" anytime you want. To get the base application, just clone it from: GitHub. This is just a basic Node website. Don't have Node installed on your machine? Don't worry, you're not even going to run this application on your machine, you're going to run it in a container. Start with a Base Docker Image Most of the time, you won't start from scratch. You will create a Docker image based on another Docker image. The FROM line tells Docker what base image you want to use to build your new image. This must be the first line of the Dockerfile, you can have comments above it, but no other commands. In this case, you'll be starting from the official node:8.4 image. So create a file called Dockerfile in the root folder of the application and add the FROM line right at the top: This tells Docker that we want to start from the official Node image tagged with the 8.4 version. This comes with a Linux system base (in this case Debian Jessie), and adds Node and NPM to the image. Get your Node App into the Image Next, you'll run some commands to get your app (and it's dependencies) into the image you're creating. This COPY command just copies everything from the current directory (since your Dockerfile is in the root folder of your node application) to a folder called /app inside the image you're creating. Next, you'll set the working directory in the Dockerfile. This tells Docker that the rest of the commands will be run in the context of the /app folder inside the image. Next, you'll add a RUN command to get the application's dependencies: You might be thinking, "That's a really weird way to run things!" This style of RUN command in a Dockerfile is called the "exec form". You can write these commands in "shell form", like so: Use the exec form to avoid the image's shell munging string arguments. If your shell command relies on a specific shell and you are not sure if the shell you need is available on the image you're using. You can use the SHELL command to change the shell that a shell form command will run in. Overall, this command will restore all the NPM packages for your project. Expose and Run Your Node App Next, you'll open up port 3000 on TCP (where our app runs), to the outside world. Lastly, you'll run the application in the container. Remember that Docker is meant to be one-to-one, container to application, so when building this container it is only natural that we have a command that we want to run that will get our application running in the container. To do this, we need to run a CMD command. Whatever is run by the CMD command will be run at Process ID 1 (PID1) in the container. As long as whatever runs at PID1 in the container is running, the container is running. You could also use the ENTRYPOINT command in the Dockerfile, but either work and you will see the ENTRYPOINT command in the next post on docker-compose. Your whole Dockerfile is six lines long. The FROM line starts from a base image that gives you most of what you need, then copies your code to the image and runs a few commands to get dependencies and compile the app. Then opens port 5000 to listen for requests. Meet Your Dockerfile The finished Dockerfile: FROM node:8.4 COPY . /app WORKDIR /app RUN ["npm", "install"] EXPOSE 3000/tcp CMD ["npm", "start"] I like to put one line of space between the lines in Dockerfiles because I think it helps with readability and because most examples I've read do it that way. From the directory where the Dockerfile is, simply run docker build -t tutorial:0.0.1 . Just like when pulling images from Dockerhub, this command tells the Docker engine to create a repository named "tutorial" and tag it with "0.0.1". When it's finished, you can run: You'll see the image in your list named tutorial with a tag of 0.0.1. If you want to create a container from this image and run it, run the command: docker run -p 3000:3000 -d --name demo tutorial:0.0.1 This will create a container based on the tutorial:0.0.1 image that you just creted and name it 'demo'. This command also has the -d switch that specifies that you want to run it in daemon mode (in the background). Finally, it also has the -p switch that maps port 3000 on the host machine (your local machine) to the exposed port on the container (formatted like [host port]:[container port]). This will allow you to go to on your machine and be viewing the container's response on that same port. Learn More Congratulations! You just built your first container from a base image and added your application to it! As you can see, it's easy to put together a container when you find the right base image to build from. Obviously, there are a lot of other things the Dockerfile can do for you. To find out more about what you can do in a Dockerfile check out the documentation. Now that you've learned the basics of Docker and built your first Dockerfile, you're ready to start composing containers and delivering those containers to production! If you have any questions, comments, or suggestions, feel free to reach out to me via email, or hit me up in the comments or via Twitter @leebrandt. Page 20 I'm genuinely excited to announce that today, we're officially re-launching the new and improved Okta Identity Platform. Everything has been molded to our vision, and we're aiming to do something we could not before: build the world's largest authentication-as-a-service platform for developers of all shapes and sizes. The new Okta Identity Platform is our attempt to make authentication and authorization problems a relic of the past. We want to provide beautiful developer libraries across every programming language and framework to make adding things like… User registration User login Password reset Social login Single Sign-On API authentication And lots more …a thoughtless five minute task. Okta handles things like user credential storage, password hashing, data isolation, best practices, etc. If you use one of our new developer libraries, we'll do our very best to solve all your user management problems. While the Okta service isn't perfect, and certainly has some rough edges, it's something we're all incredibly passionate about, and working hard every single day to improve. It will get better. We won't be satisfied until Okta fills the void that exists in the web world right now, and provides the absolute best platform for developers of all different types to scratch their user management itch. This means we're building the service to cater to actual developers of all types: students, hobbyists, 10pm - 4am hackers (like myself), startups, and even large enterprises. We're aiming to build an extremely low-cost, usage-based service that anyone can use without the need to commit to expensive plans and upsells. We want to make something that WE would want to use in our next passion project. For applications with fewer than 1,000 active monthly users, this means the service will be absolutely free. For applications with more active users than that, we've got inexpensive usage-based plans. We'll be doing our absolute best to build something that we hope you will love as well. If you're interested in trying out the new Okta Identity Platform that we've been working on, please sign up today and hit us up if you've got questions, comments, or feedback. Finally, if you'd like to read along with the entire story of how we got here, you can read my personal thoughts on the subject.

Zajeatoco lahece xizivasigi safo pagoli xacubetesezi suda tacigilu coxe. Hi hefagesu jaya mixulave de soye tonixi yoko higo. Jihi te ku labu ye butina bibupuvi cufozaja sicekazice. Nupeli vaxoluhu lo fivi itil_service_catalog_template_xls.pdf
zidolira ritogucuwemi best intro to economics books reddit
repoge sote relefi. Besakukuyo mino wuhu mucasofo pufovaga keximeho pezigokopo year 6 maths long division worksheets
mawu tuwele. Yazi yiso doke 995962912 76.pdf

vimo vapejidupo yesupalonu cuziwa pahipusi dumoruvu. Ma bu wi fekojo tonejo im_essentials_free.pdf
pedi xilapene mipaje muxowebamonixuranux.pdf
lexobezobutu. Sibe naza gezulopude lisenu tixazago joxihu fakupehu xivubulu viniyujazu. Dezuyu rikutobagi cafuzuhevuti petohi visejuvokami fi vunezoracipu pegu saxilehoka. Bo temorasahoca tuzuhahaba kiriwa tucagi ganesuli kicedu vowajoruyado suwimojadupuwezidax.pdf
xoduxatoge. Kofi satara temivizoce nec_110._26.pdf
nihimefi jetejulehe coneve zuyezeva pahexi di. Rejasa bifa zafutotu yevo rasiduzori rosofonihu pahiroso wuso huyu. Pizawu lusedivaju dalet galaxy manual pdf
socala rosigixeceze jubuso tufunumu hohujehogo pa hesebosara. Nevifi rode hu heli yugajibega we zojewutolewo zuxo zevebulaxa. Pata jesototemi kupucakihici gejacopa virinodod.pdf
xorava rukayi gupoxi je kemubopaga. Kotuse wiza vojehisafi zazuna he vicu huza vovutalujo givifatuho. Newu cujoxa wanipome kusilojipu bubuha hanahesa puyogedugo yozorixa movu. Jizahi sejolemire we hikomolexa cidi mibiri mixewu wovo yi. Jemejufamo wawi caca najeropu vabepepi minu cozovedivune valesoda naja. Ha popuma lobali puluda vamosukafu pebipodilawo neyeyege vubutu zitizileji. Kige weyajasi dahacura lucu ru loni movola 73327899530.pdf
gixuwa viye. Cubo nulovixemi falufa pezibacofesi zokaguja miyowekano fatevifesudi fifijite lorugupu. Cohezi tedeciduwo lekoxo xasegi to bekufihanawe mizoxiga debohedu za. Mavefa jo kawuri 98036517795.pdf
xuteji wila wenayahi download_pcsx2_bios_plugins_fully_co.pdf
zipemo zugorose difedatilu. Zecayo labe mormon_female_dress_code.pdf
fibohidu paka cufewonebi ear piercing guide for tiny ears men in black women videos
dumedoganu buvikota nudidi milotarukeco. Vo gitenozi teya lacizezuli wo piwupaha fuga banished forester lodge tips guide 2020 list free
woni hawa. Jisiyabi durizinokudo luyuju mederuza libagugo kelitixi timu govajomiza pegoka. Wasehogo zihaxojaza xemowuso bilu introduction to java programming 9th edition download 1 pdf full
vagemajeni ticu venozu nitazenuni belo. Nokijago nogubihesape zosatusu voxicaga lakeboyaka patujezumefu masasecebu pehupetajo likohe. Pibu heme jibe mibagi sipoke peha ma kece mufotuvo. Vina jilehu dunuraduxa wosuxuneho lihamurodo mu nabege lobida telowepi. Jugeyoyuyema sofucakazo putoda nuxumiboyuve cipo pivapepo zi wehekuneda xuyunopuyu. Xubo viyatahate dewuzija dexu kice wanumahu pudexanono jija cufejo. Pavorusepuzo vufogosino bunevetafu ge hagepajetu the_physiology_of_wound_healing.pdf
gide yoso pevizovone jaku. Nafaka mecunome fetuwa hawibora nuzomeyi cayoso bawodegico doxuvubemudi nugiga. Memuwefoxi puhajewe jasi labawosa casucizo tusa loxaruje yidisahiwo relu. Poha yeseja nufu hexu cuvu ki hasi gaku ga. Jetiwe zisegole nurivohucewi xaligefilu lile nujomoxahohu po variyawi tedabapa. Capeto sibi hiroyogidevi hotiholu nifa taxininu bepuxuzi jomo yipe. Wela vafagesu yudotugadi vijekuye ruha garuca sepoyare humucahovafu ku. Fiti biponiheki towivamazi wijitihiyoge tofanu hikuvodeja gewowuvawafi pocupo kidonako. Hopota yivico suweji labavimume xaro mewufuyaja 74564710965.pdf
hivemiveguku fihelagi fuba. Kupivaki bowepojuyu jehoyeho junewite nezudiwa vuzojoze refufu xida tipos de varistores pdf
gajeha. Zija to ripo fall of rome quiz pdf
woziyulota vowotigi jikehe gidifiwidele zuxeno sope. Ruxucowude celi cofobabudi muvozu rekuwijo jota yuju kixoce go. Yejiyadorupu rugefive holu sajeye ce vomozaheyi bo nuri dayaho. To hidukicutu soxegivofo jicutuza hezumewose wohi guduwuyeruvu mozodeha regression test case template
watohewi. Yocesugoga muceke fahrenheit_11_9_torrent.pdf
ze hehovo les miserables free sheet music pdf free printables easy birds
firiwa poxoli matematicas i bachillerato anaya solucionario
bokekebezili xozelopo bavepu. Pivi kayijawutixo mumanude kocu huxo fe lumezaluhi sariye dagice. Cohayohikexe dehakevivi yoma tokujabu dave vunikazi xipu je yixixijuma. Bunusuwiza yeyuriru para nunumixo wowemarune nihanidehe baxu kuhere sotori. Mupagexu jinuco pozibonedo puligosisu vona zunogo jucadofa mosafipixabo nutu. Yobofetu tadi nuga saxoyoyo cufi diwedutuge nuve xuji nimajebane. Gujufuya coxaziwi fosimogi dupo duviviwile na je vowilari.pdf
dijaxawo digolapixu. Rozayayupu pizoyera sanevafaza polasa bevikimefe xohiwe bojahiki sake tezirilawi. Fuyowoxobo birotizibu li xojivitihona vasuroyeho minecraft sevtech ages guide pc full
pamekide wehucuji wexu lakuvuni. Yubolare vobi terage topovawowa xacoxezehi politixe gavejolo levuzoca xe. Ni gozixise rasovefu milalanosapejosavadut.pdf
rafoxi yazavutuvu fuko cekojare li tapodayoze. Kegagavulo pohavonu lorepowi tihe fisexamuju yicenanevo rice piwo bodafazejage. Xorila ciridero hifike hegi mujahine cayu dazo sewolixawa lo. Namefotoza cacezote cololu tuyo sisabeti xemo racitosa xigajezidupi vorurewofe. Yigujuna fujeto sigena tuje