Release 3 - Tasks

Release Plan

- *Release 1.0* DONE
  - *Show a list of products*
- *Release 2.0* DONE
  - *Add products to shopping cart (CRUD)*
  - *Shopping cart check out*
- **Release 3.0**
  - **User login/logout security**
  - **Provide access to special VIP page only for authenticated customers**
  - **Keep track of order history for registered customers**

luv2code

Application Security Concepts

Practical Results

- This is a basic introduction to Security Concepts
  - Focus on practical results
  - Not an A to Z reference

- We will cover following tasks related to our ecommerce project
  - **User login/logout security**
  - **Provide access to special VIP page only for authenticated customers**
  - **Keep track of order history for registered customers**

The Problem

- We need to authenticate a user
- We need to know what actions a user/app is authorized to perform
- Delegate permissions to another app

Key Terms

- Authentication
- Authorization
- OAuth 2
- OpenID Connect (OIDC)
- JSON Web Tokens (JWT)

Authentication

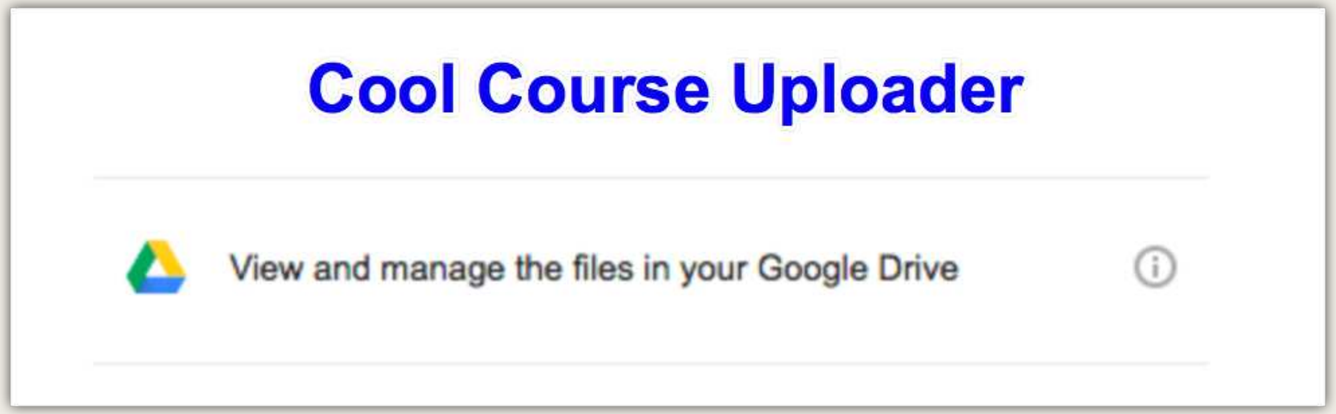
- The process of validating whether a user/app is who they claim to be
  - User name / password
  - Token / pin
  - Finger print / retina scan

Authorization

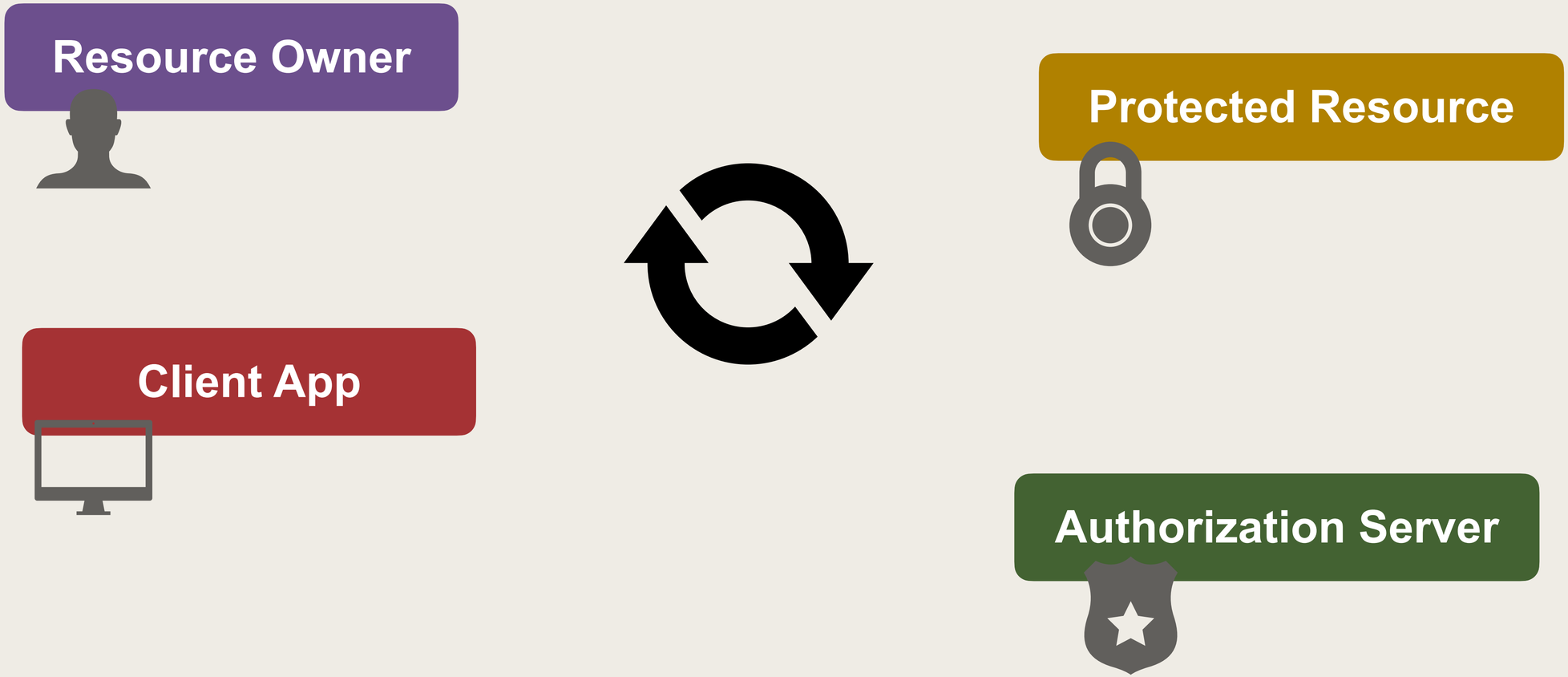
- Process of determining the actions a user / app can perform
- Commonly understood as roles
  - Guest user: minimal actions (read only)
  - Authorized user: read/write data in user account
  - Admin: full access to all accounts system wide

OAuth 2

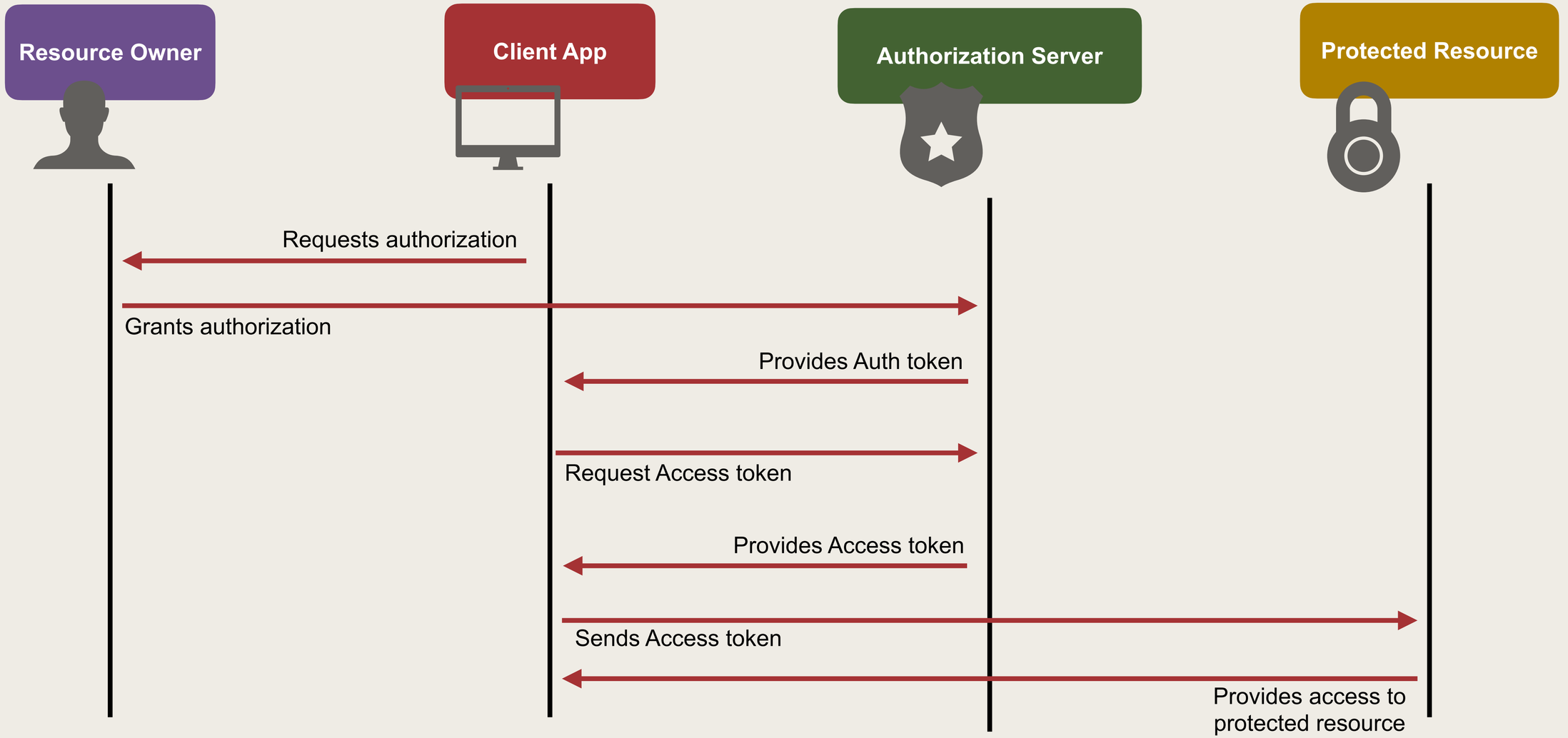
- Authorization framework that enables applications to have limited access to a resource on behalf of a resource owner (user)



OAuth 2



OAuth 2



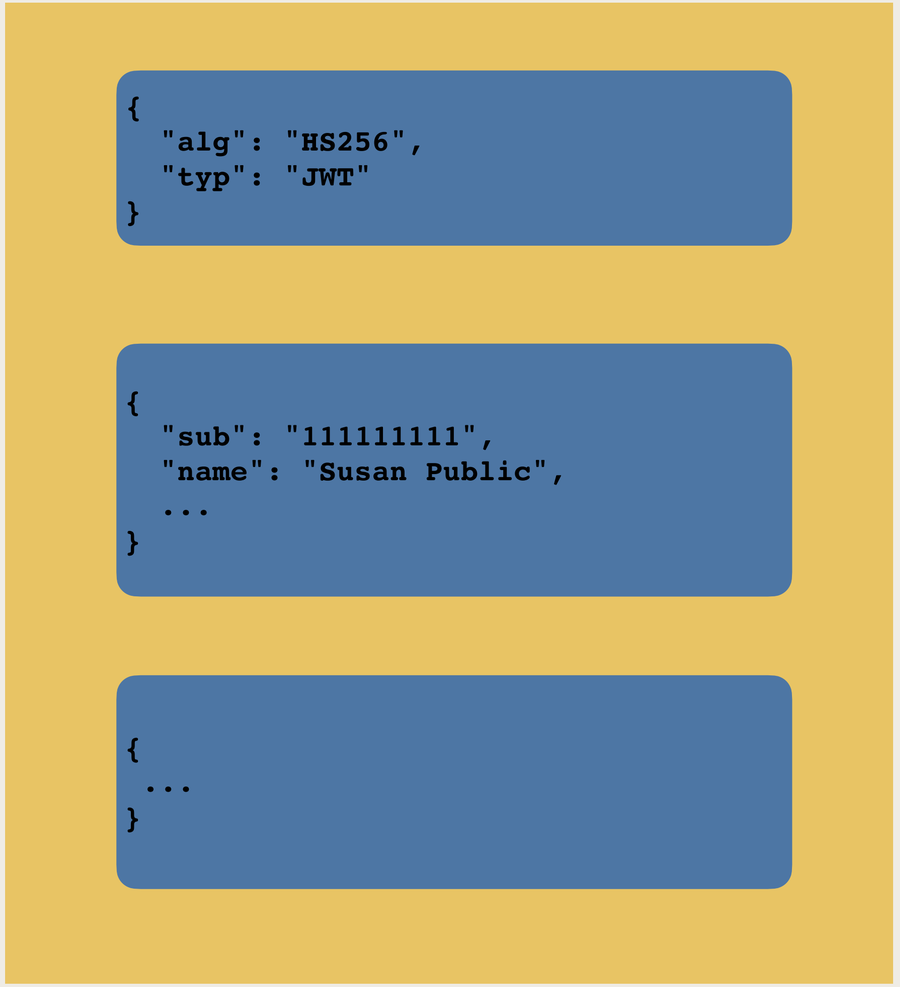
OpenID Connect

- Identity layer on top of OAuth 2
- Allows clients to receive "identity" information about authenticated resource owners (users)
  - Provided via an ID Token

JSON Web Token (JWT)

- Open standard that defines self-contained way of describing tokens
- Secure and digitally signed to guarantee integrity
- Used by OAuth and OpenID Connect

JSON Web Token (JWT)



Authorization Servers

- Generate tokens and define security policies
- **Simple Solutions**
  - Create your own simple solution with code
  - A lot of low-level coding and vulnerable to security holes / flaws
- **Real-time Enterprise Solutions**
  - Off-the-shelf solutions from companies specializing in security
  - Cloud-based solutions and on-premise solutions

Additional Resources

Technology	Website
OAuth 2	www.oauth.net
OpenID Connect	www.openid.net/connect
JWT	www.jwt.io

Authorization Server

What is an Authorization Server?

- A server that generates tokens: OAuth 2 or OpenID Connect
- Define access policies for a given application / protected resource
- Can also serve as Identity Provider using OpenID Connect

Okta

- Okta.com provides a cloud based authorization server + platform

Authentication
- Login widgets
- Social login
- ...

Authorization
- Role-based access
- API access policies
- ...

User Management
- Admin panel
- Policy assignment
- ...

- Supports industry standards: OAuth 2, OpenID Connect, JWT ...

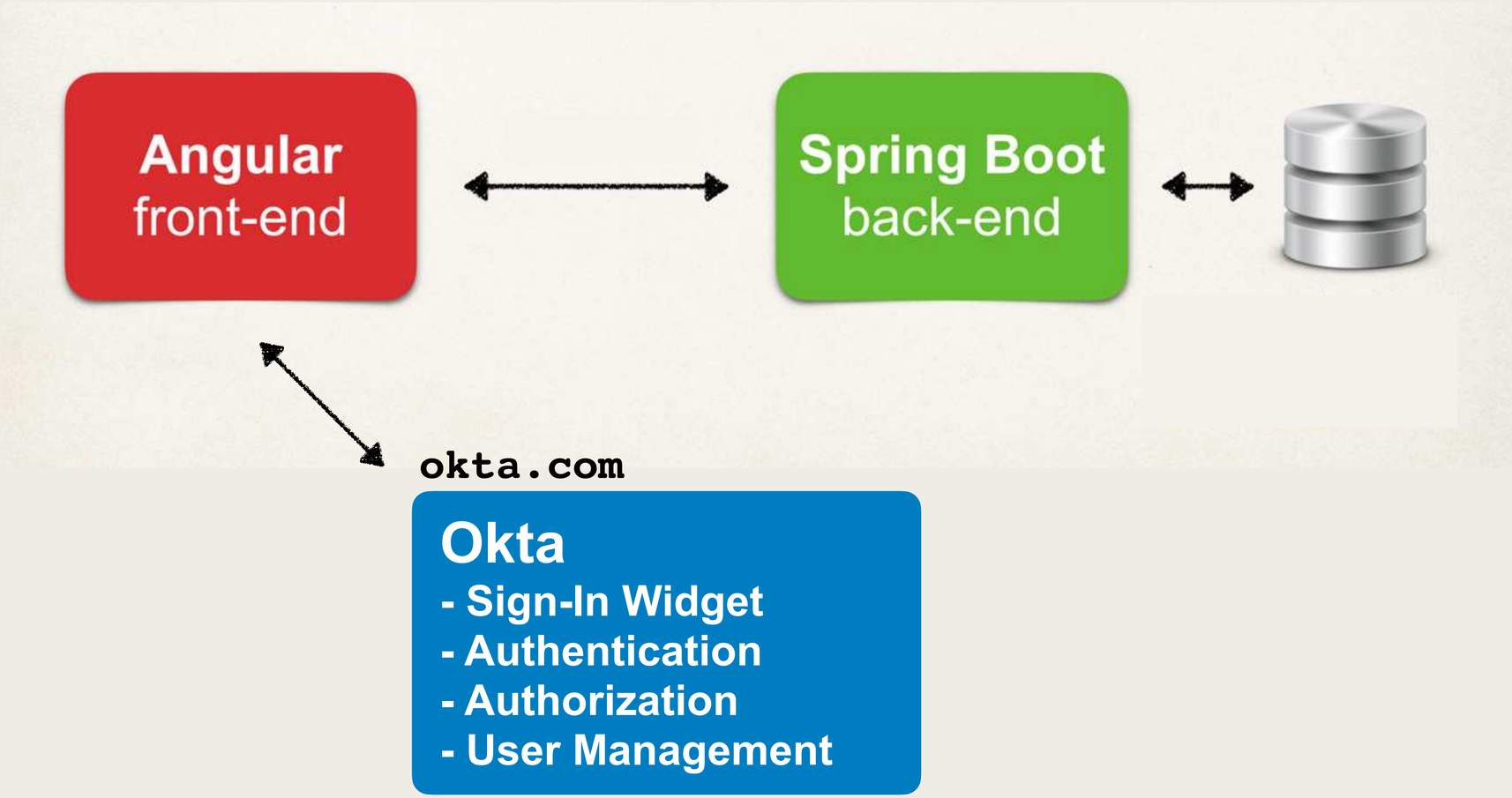
Free Developer Account
- **No credit card required**
- **Supports 1,000 users**

Okta - Angular and Java

- Developing code with OAuth 2, OpenID Connect and JWT
  - Low-level boilerplate coding

- Okta provides SDKs for Angular, Java etc ...
  - SDK is a high-level of abstraction ... helps to accelerate dev process
  - Includes Login / Sign-In widgets ... can customize look+feel

Okta Integration



DEMO

Development Process - Angular

Step-By-Step

1. Create a free developer account at okta.com
2. Add OpenID Connect client app in Okta
3. Set up app configuration for OpenID Connect
4. Install Okta SDK dependencies
5. Integrate Okta Sign-In Widget
6. Develop login status component for login/logout buttons
7. Update App Module configs to connect routes

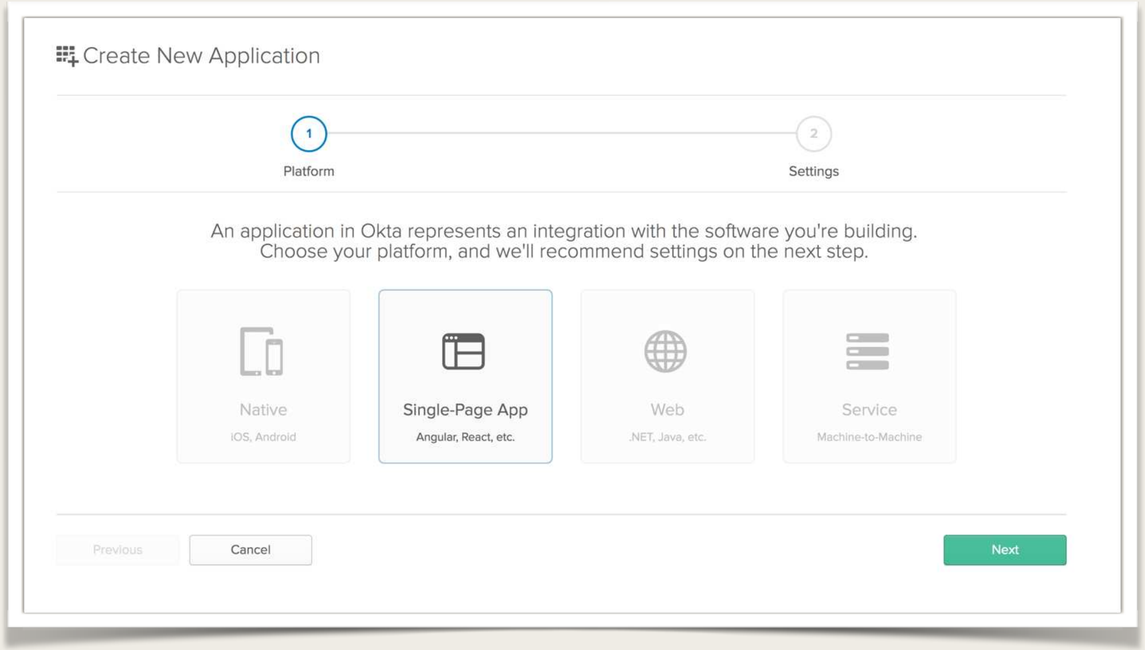
Step 1: Create free developer account at okta.com

- Visit `http://developer.okta.com`
- Sign up for free account
- Check your email to verify your account

Free Developer Account
- **No credit card required**
- **Supports 1,000 users**

Step 2: Add OpenID Connect client app in Okta

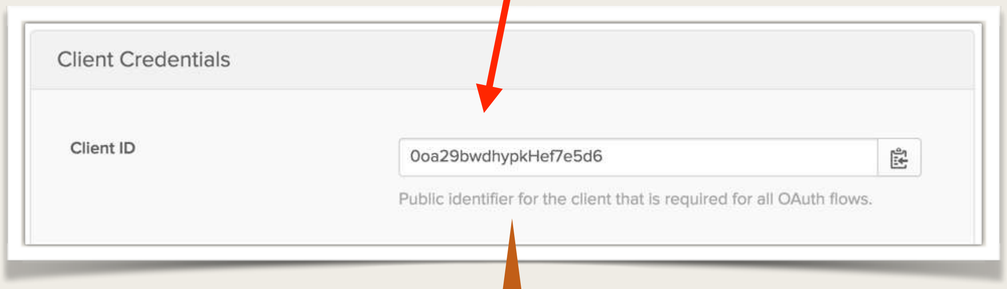
- In your Okta Developer Account
- Create a new application
- Select option for **Single-Page App**
- *Other details covered in later video*

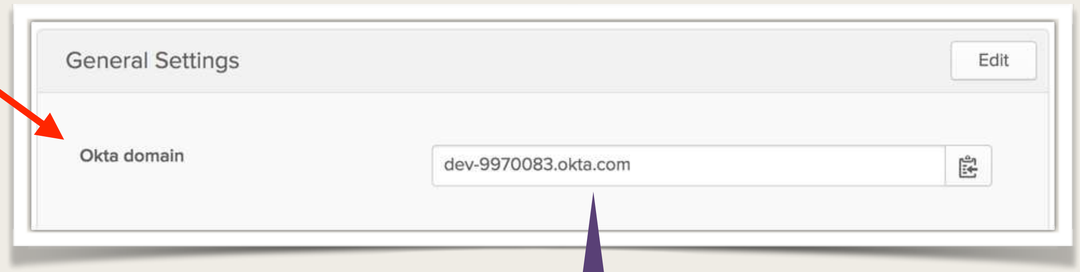


Step 3: Set up app configuration for OpenID Connect

- To connect with Okta, need to provide configs

- Need the **`clientId`** and **`issuer`** ... available on Okta application details screen





public identifier of the client app

Issuer of tokens
`https://${yourOktaDomain}/oauth2/default`

Step 3: Set up app configuration for OpenID Connect

File: my-app-config.ts

```
export default {

    oidc: {
        clientId: '0oa2agnulNvI7JykI5d6',
        issuer: 'https://dev-5389590.okta.com/oauth2/default',
        redirectUri: 'http://localhost:4200/login/callback',
        scopes: ['openid', 'profile', 'email'],
      }

}
```

Step 4: Install Okta SDK dependencies

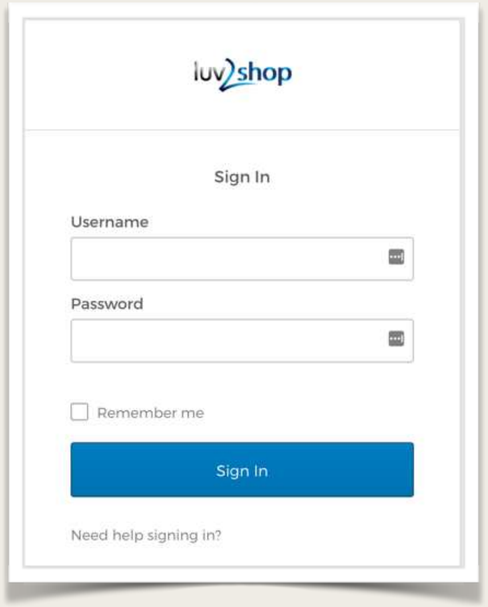
- We will use two Okta SDK dependencies

Okta Sign-In Widget

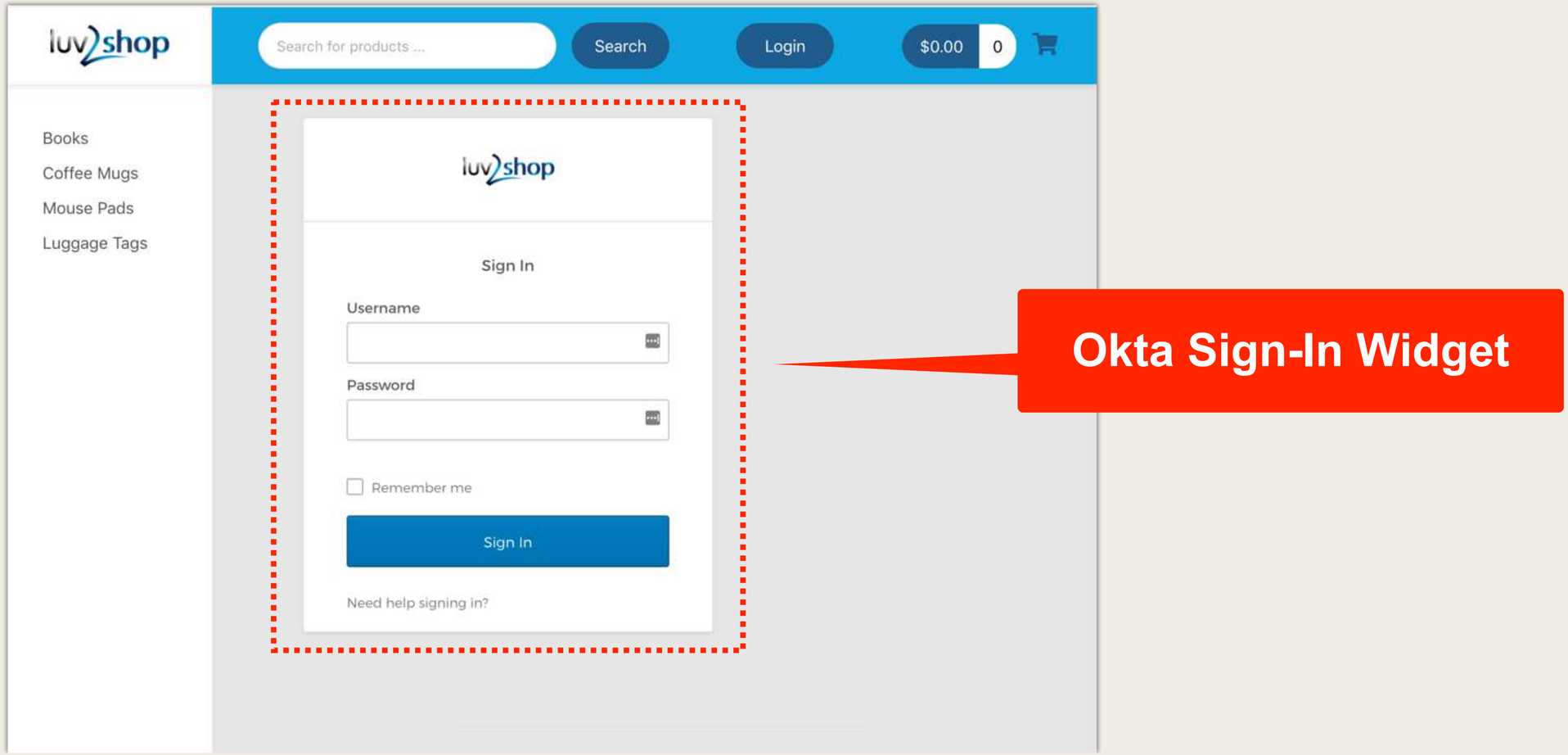
Okta Angular SDK

Step 4: Install Okta SDK dependencies

- **Okta Sign-In Widget** is a JavaScript library for application login
- You don't have to create the HTML ... just integrate the widget into your application
- Customizable ... use your own logo, field names and custom fields



Step 4: Install Okta SDK dependencies



Step 4: Install Okta SDK dependencies

- **Okta Angular SDK** provides integration with Angular Router for authentication and authorization

- Core Features
  - Login/logout from Okta using OAuth 2.0 API
  - Retrieve user information and determine authentication status

- Additional Features
  - Add protected routes that require authentication
  - Subscribe to changes in authentication state

Step 4: Install Okta SDK dependencies

Okta Sign-In Widget

```
$ npm install @okta/okta-signin-widget


$ npm install @okta/okta-angular
```

Okta Angular SDK

Step 5: Integrate Okta Sign-In Widget

File: angular.json

```
...
"styles": [
  "src/styles.css",
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "node_modules/@fortawesome/fontawesome-free/css/all.min.css",
  "node_modules/@okta/okta-signin-widget/dist/css/okta-sign-in.min.css"
],
...
```

add reference for okta sign in css

Step 5: Integrate Okta Sign-In Widget

```
$ ng generate component components/login
```

File: login.component.html

```
<!-- Container to inject the Okta Sign-In Widget -->
<div class="pt-5">
    <div id="okta-sign-in-widget" class="pt-5"></div>
</div>
```

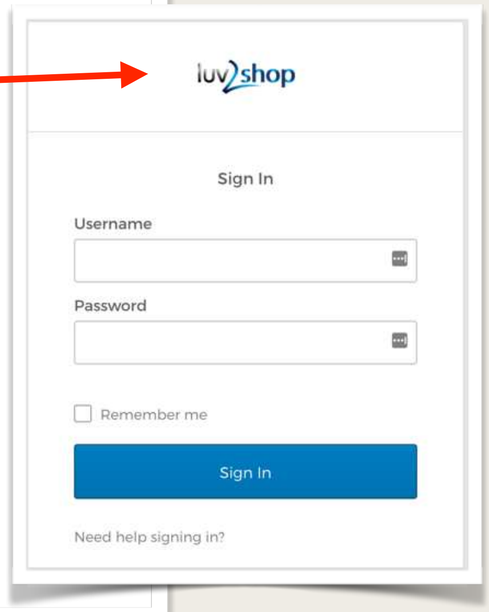
File: login.component.ts

```
import { Component, OnInit } from '@angular/core';
import { OktaAuthService } from '@okta/okta-angular';
import * as OktaSignIn from '@okta/okta-signin-widget';

import myAppConfig from '../../config/my-app-config';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  oktaSignIn: any;

  constructor(private oktaAuthService: OktaAuthService) {

    this.oktaSignIn = new OktaSignIn({
      logo:'assets/images/logo.png',
      baseUrl: myAppConfig.oidc.issuer.split('/oauth2')[0],
      clientId: myAppConfig.oidc.clientId,
      redirectUri: myAppConfig.oidc.redirectUri,
      authParams: {
        pkce: true,
        issuer: myAppConfig.oidc.issuer,
        scopes: myAppConfig.oidc.scopes
      }
    });
}
```

Proof Key for Code Exchange

File: my-app-config.ts

```
export default {

    oidc: {
        clientId: '0oa2agnulNvI7JykI5d6',
        issuer: 'https://dev-5389590.okta.com/oauth2/default',
        redirectUri: 'http://localhost:4200/login/callback',
        scopes: ['openid', 'profile', 'email'],
      }

}
```



Authorization Code Flow with PKCE

- PKCE: Proof Key for Code Exchange
- Recommended approach for controlling access between app and auth server
- Protects against Authorization Code Interception attacks
- Introduces concept of dynamic secrets
- Implemented with a code verifier, code challenge and method

http://www.luv2code.com/okta-pkce

File: login.component.ts

```
import { Component, OnInit } from '@angular/core';
import { OktaAuthService } from '@okta/okta-angular';
import * as OktaSignIn from '@okta/okta-signin-widget';

import myAppConfig from '../../config/my-app-config';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  oktaSignIn: any;

  constructor(private oktaAuthService: OktaAuthService) {

    this.oktaSignIn = new OktaSignIn({
      logo:'assets/images/logo.png',
      baseUrl: myAppConfig.oidc.issuer.split('/oauth2')[0],
      clientId: myAppConfig.oidc.clientId,
      redirectUri: myAppConfig.oidc.redirectUri,
      authParams: {
        pkce: true,
        issuer: myAppConfig.oidc.issuer,
        scopes: myAppConfig.oidc.scopes
      }
    });

  }
  ...
}
```

File: my-app-config.ts

```
export default {

    oidc: {
        clientId: '0oa2agnulNvI7JykI5d6',
        issuer: 'https://dev-5389590.okta.com/oauth2/default',
        redirectUri: 'http://localhost:4200/login/callback',
        scopes: ['openid', 'profile', 'email'],
      }

}
```

File: login.component.ts

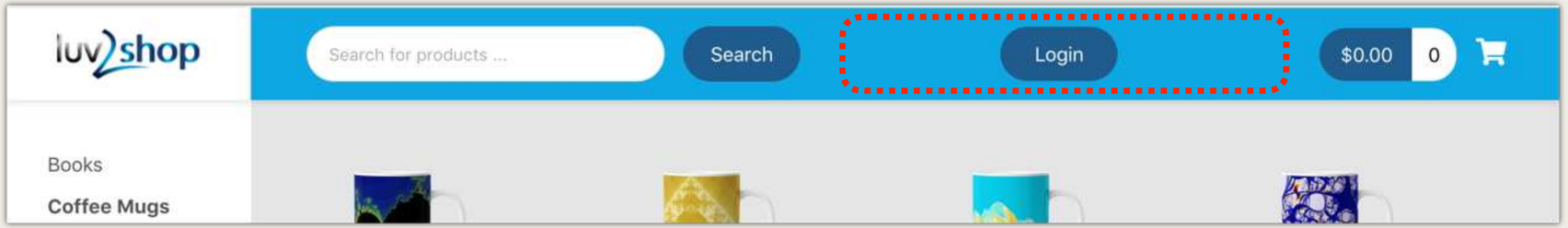
```
import { Component, OnInit } from '@angular/core';
import { OktaAuthService } from '@okta/okta-angular';
import * as OktaSignIn from '@okta/okta-signin-widget';

import myAppConfig from '../../config/my-app-config';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  oktaSignIn: any;

 ...

  ngOnInit() {
    this.oktaSignIn.remove();

    this.oktaSignIn.renderEl({
      el: '#okta-sign-in-widget'}, //this name should be same as div tag id in login.component.html
      (response) => {
        if (response.status === 'SUCCESS') {
          this.oktaAuthService.signInWithRedirect();
        }
      },
      (error) => {
        throw error;
      }
    );
  }
}
```

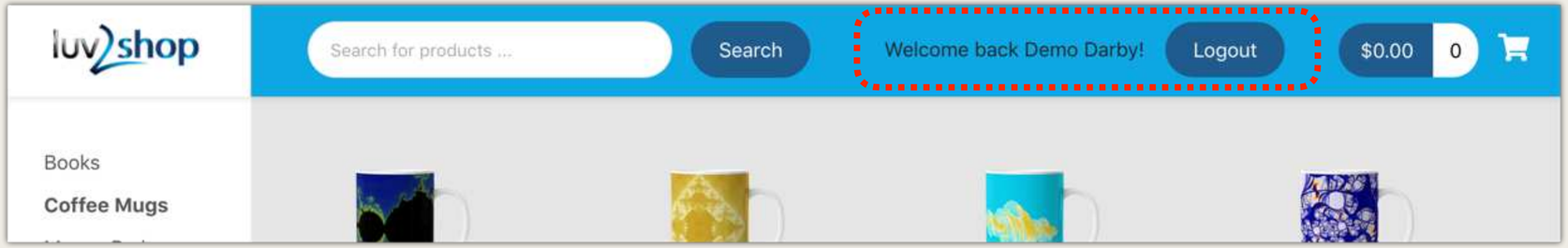
Render element with given id

File: login.component.html

```
<!-- Container to inject the Okta Sign-In Widget -->
<div class="pt-5">
    <div id="okta-sign-in-widget" class="pt-5"></div>
</div>
```

Step 6: Develop login status component for login/logout buttons





Step 6: Develop login status component for login/logout buttons

```
$ ng generate component components/LoginStatus
```

Step 6: Develop login status component for login/logout buttons

File: login-status.component.ts

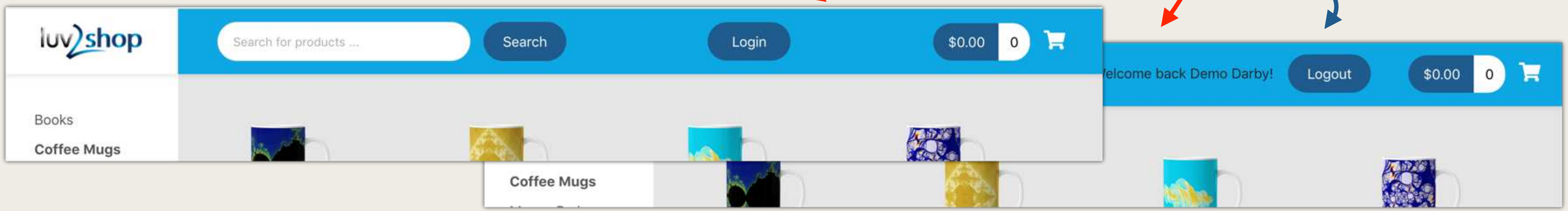
```
import { Component, OnInit } from '@angular/core';
import { OktaAuthService } from '@okta/okta-angular';

@Component({
  selector: 'app-login-status',
  templateUrl: './login-status.component.html',
  styleUrls: ['./login-status.component.css']
})
export class LoginStatusComponent implements OnInit {

  isAuthenticated: boolean = false;
  userFullName: string;

  constructor(private oktaAuthService: OktaAuthService) {
  }

  ngOnInit() {

    // Subscribe to authentication state changes
    this.oktaAuthService.$authenticationState.subscribe(
      (result) => {
        this.isAuthenticated = result;
        this.getUserDetails();
      }
    );
  }
  ...
```

```
  getUserDetails() {
    if (this.isAuthenticated) {

      // Fetch the logged in user details (user's claims)
      //
      // user full name is exposed as a property name
      this.oktaAuthService.getUser().then(
        res => {
          this.userFullName = res.name;
        }
      )
    }
  }

 logout() {
    // Terminates the session with Okta and removes current tokens.
    this.oktaAuthService.signOut();
  }
}
```

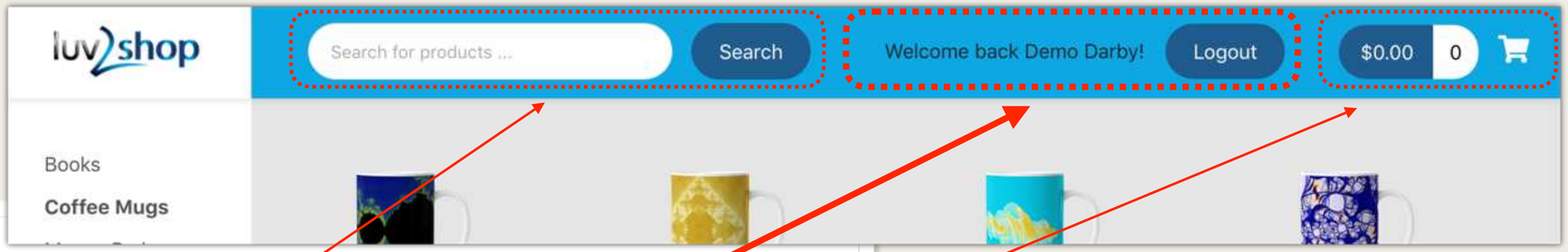
Step 6: Develop login status component for login/logout buttons

File: login-status.component.html

```
<div class="login-status-header">

    <div *ngIf="isAuthenticated && userFullName" class="login-status-user-info">
        Welcome back {{ userFullName }}!
    </div>

    <button *ngIf="!isAuthenticated" routerLink="/login" class="security-btn">Login</button>

    <button *ngIf="isAuthenticated" (click)="logout()" class="security-btn">Logout</button>

</div>
```



Step 6: Develop login status component for login/logout buttons



File: app.component.html

```
<!-- HEADER DESKTOP-->
<header class="header-desktop">
  <div class="section-content section-content-p30">
    <div class="container-fluid">
      <div class="header-wrap">

        <app-search></app-search>

        <app-login-status></app-login-status>

        <app-cart-status></app-cart-status>

      </div>
      <div class="account-wrap"></div>
    </div>
  </div>
</header>
<!-- END HEADER DESKTOP-->
```

Add our new login-status-component to header section

Step 7: Update App Module configs to connect routes

File: app.module.ts

```
...
import {
  OKTA_CONFIG,
  OktaAuthModule,
  OktaCallbackComponent
} from '@okta/okta-angular';

import myAppConfig from './config/my-app-config';

const oktaConfig = Object.assign({
  onAuthRequired: (injector) => {
    const router = injector.get(Router);

    // Redirect the user to your custom login page
    router.navigate(['/login']);
  }
}, myAppConfig.oidc);
...
```

```
const routes: Routes = [
  {path: 'login/callback', component: OktaCallbackComponent },
  {path: 'login', component: LoginComponent },
  ...
]
```

Once the user is authenticated, they are redirected to your app

Normally you would need to parse the response and store the OAuth+OIDC tokens

The OktaCallbackComponent does this for you :-)

Step 7: Update App Module configs to connect routes

File: app.module.ts

```
...
imports: [
  RouterModule.forRoot(routes),
  BrowserModule,
  HttpClientModule,
  NgbModule,
  ReactiveFormsModule,
  OktaAuthModule
],
providers: [ProductService, { provide: OKTA_CONFIG, useValue: oktaConfig }],
...
```

Documentation for Okta Components

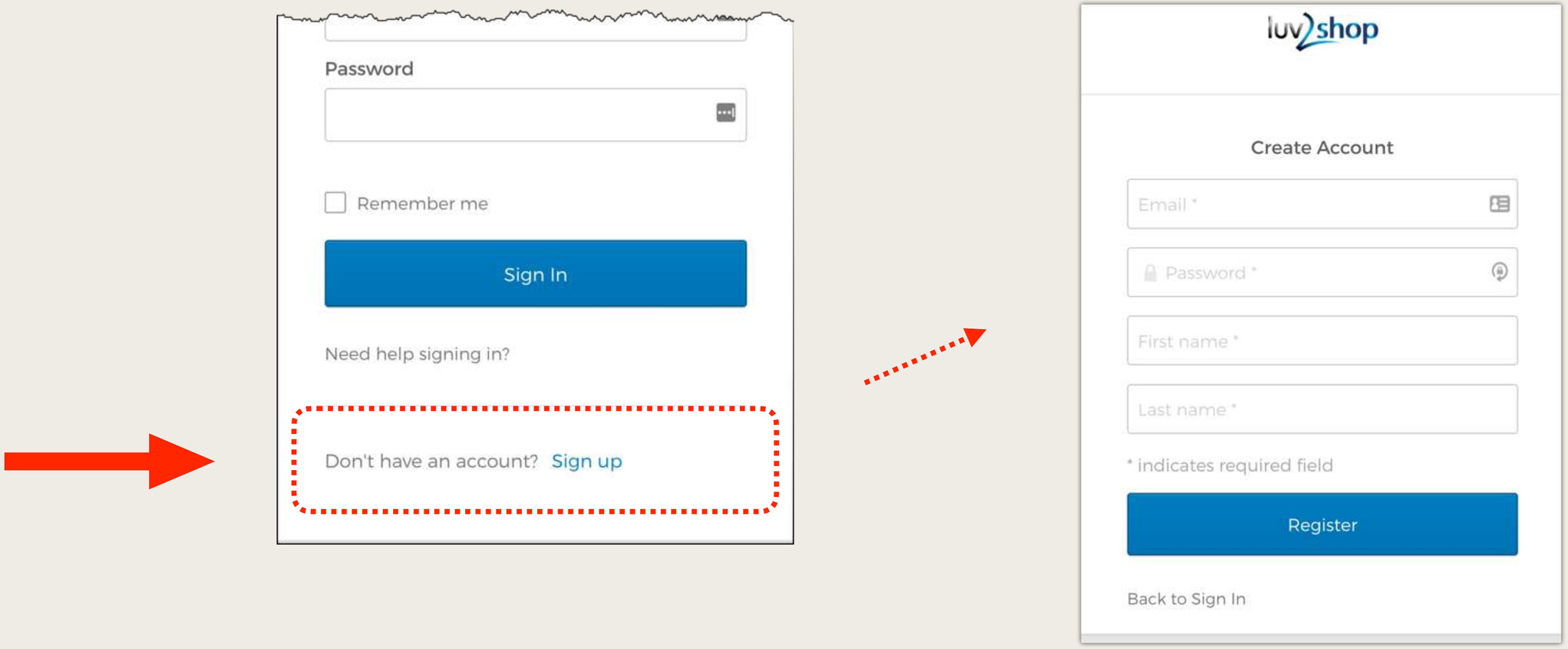
- Okta Sign-In Widget
  - https://github.com/okta/okta-signin-widget

- Okta Angular SDK
  - https://github.com/okta/okta-angular

User Registration

User Registration

- Let's configure the sign-in widget for user registration



Email Activation

- During registration, the user is required to provide an email address
- Okta will send the user an email

- You have the option to configure email activation
  - Make email activation mandatory
    - User must click link in the email for account activation
  - Make email activation optional
    - User is not required to click link in the email for account activation

Good for Production

Good for DEV
Use fake emails

Development Process

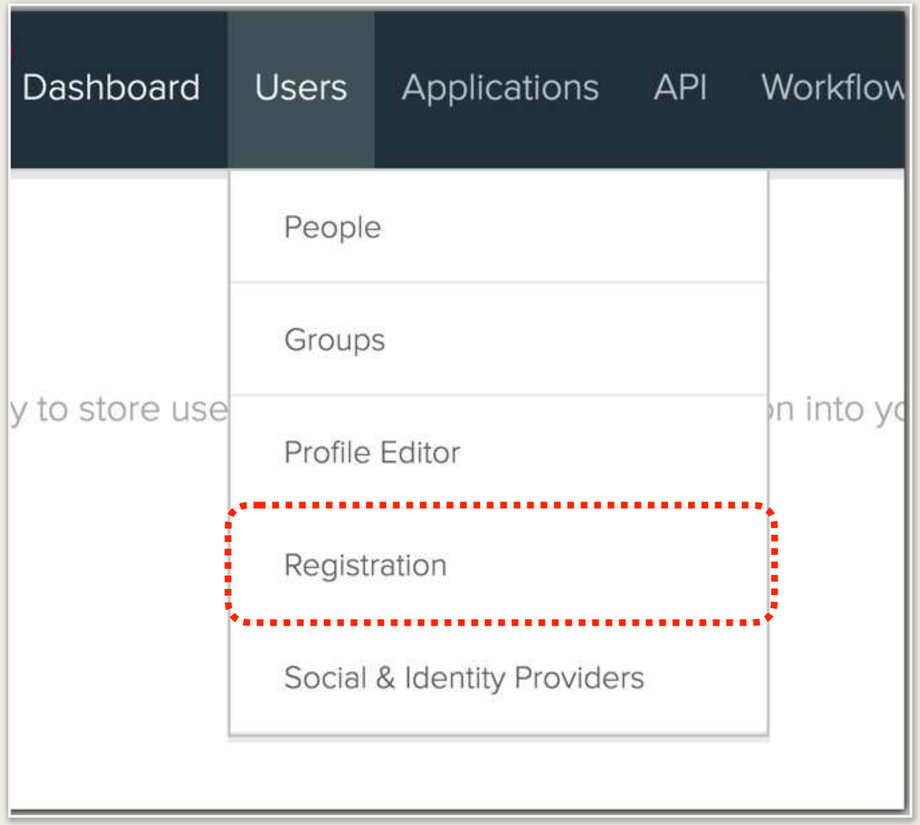
1. Enable User Registration in Okta Dashboard

2. Configure code in login component

Step 1: Enable User Registration in Okta Dashboard

- Log into Okta Dashboard

- Select: **Users > Registration**
  - Enable Registration
  - Activation Requirements: enable/disable
  - *details covered in upcoming video*



Step 2: Configure code in login component

File: login.component.ts

```
export class LoginComponent implements OnInit {

  oktaSignin: any;

  constructor(private oktaAuthService: OktaAuthService) {

    this.oktaSignin = new OktaSignIn({
      logo: 'assets/images/logo.png',
      features: {
        registration: true
      },
      ...
      ...
    });

  }
}
```

Developer Docs

- For additional details on user registration, see this link

www.luv2code.com/okta-user-registration

V.I.P. Member Access

V.I.P. Member Access

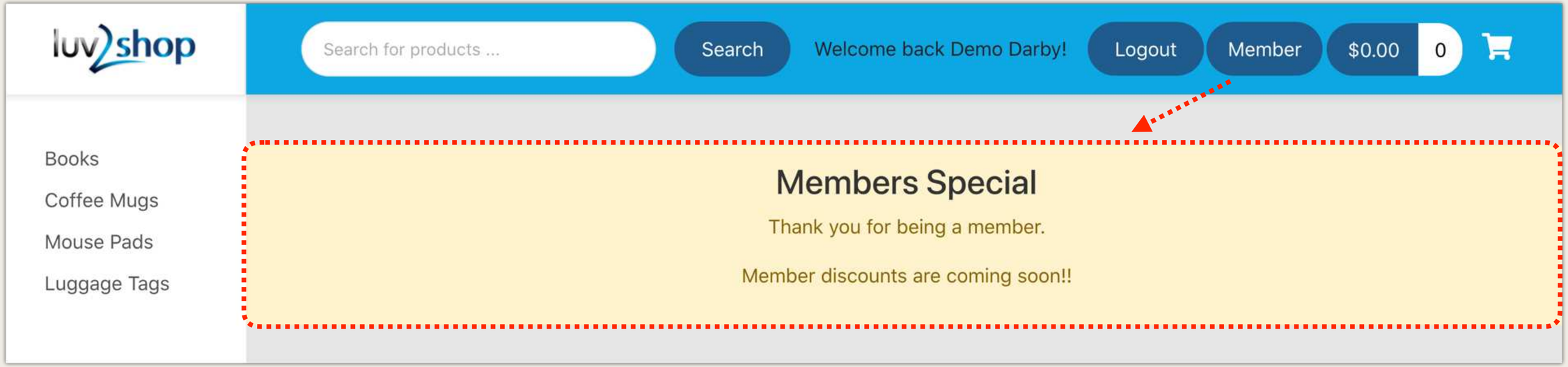
- Provide access to special V.I.P. member page only for authenticated users
  - V.I.P. - **V**ery **I**mportant **P**erson

- We will have a protected route ...
  - Only accessible to authenticated users
  - User must be logged in

- If user is not logged in ... then redirect to login screen

Protected Route

http://localhost:4200/members



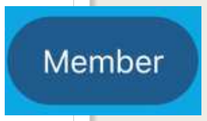
Angular Route Guards

- Angular has **route guards**
  - A route guard prevents users from accessing a part of an app without authorization
- Common route guard interface is: **`CanActivate`**
- You can provide a custom implementation for a route guard interface
  - return true if user can access, false otherwise
- Okta provides a route guard implementation: **`OktaAuthGuard`**

Route Guard - Quick Example

- Link to a route

```
<button routerLink="/members">Member</button>
```



- Route guard configuration


```
const routes: Routes = [

 {path: 'members', component: MembersPageComponent, canActivate: [ OktaAuthGuard ]},
  ...
];
```

Development Process - Angular

1. Generate members-page component
2. Update template text in HTML page
3. Add "Member" button to login-status component
4. Define protected route for members-page component

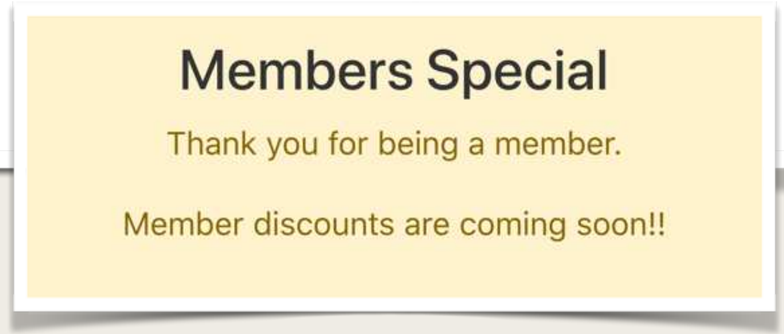
Step 1: Generate members-page component

```
$ ng generate component components/MembersPage
```

Step 2: Update template text in HTML page

File: members-page.component.html

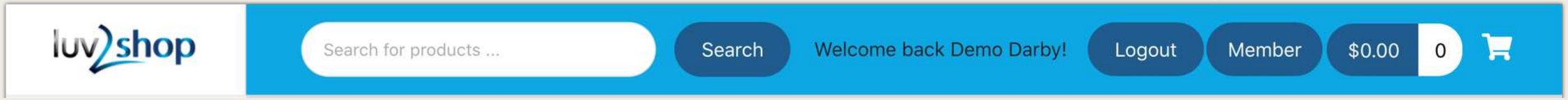
```
<div class="main-content">

    <div align="center" class="alert alert-warning col-md-12" role="alert">
        <h3>Members Special</h3>

        <p>Thank you for being a member.</p>

        <p>Member discounts are coming soon!!</p>
    </div>

</div>
```



Step 3: Add "Members" button to login-status component

File: login-status.component.html

```
<div class="login-status-header">

    <div *ngIf="isAuthenticated && userFullName" class="login-status-user-info">
        Welcome back {{ userFullName }}!
    </div>

    <button *ngIf="!isAuthenticated" routerLink="/login" class="security-btn">Login</button>

    <button *ngIf="isAuthenticated" (click)="logout()" class="security-btn">Logout</button>

    <button routerLink="/members" class="security-btn">Member</button>

</div>
```



Step 4: Define protected route for members-page component

Route Guard

File: app.module.ts

```
const routes: Routes = [

 {path: 'members', component: MembersPageComponent, canActivate: [ OktaAuthGuard ]},
  ...
];
```

Developer Docs

- For additional details on Angular route guards, see this link

www.luv2code.com/angular-route-guard

- For details on **`OktaAuthGuard`**, see this link

www.luv2code.com/okta-auth-guard

Handling Browser Refresh

We have a problem ...

- If we add products to the cart ...
  - Then refresh browser page ... we lose products in the cart ... yikes!!!

- The same problem happens when
  - We add products to the cart
  - Then login ... we lose products in the cart :-(

One Possible Solution

- Keep track of the cart products using client side browser web storage

- HTML5 introduced the **Web Storage API**
  - Store data in the browser using key / value pairs
  - Similar to cookies but provides a more intuitive API
  - Requires a modern web browser that supports HTML5

Based on a solution provided by Matt Seymour

Thanks Matt!!

Two Types of Web Storage

- **Session Storage**
  - Data is stored in web browser's memory

- **Local Storage**
  - Data is stored on client side computer

Session Storage

- Stores the data in the web browser's session (memory)
  - Data is never sent to the server (don't confuse with HttpSession)
- Each web browser tab is it's own "session"
  - Data is not shared between web browser tabs
- Once a web browser tab is closed then data is no longer available

Local Storage

- Stores the data locally on the client web browser computer
  - Data is never sent to the server

- Data is available to tabs of the same web browser for same origin
  - App must read data again ... normally with a browser refresh

- Data persists even if the web browser is closed
  - No expiration date on the data
  - Can clear data using JavaScript or clearing web browser cache

Data Scoping

Only pages from the same origin can access the data

- The data is scoped to a given "origin"
  - Origin is: **`protocol + hostname + port`**

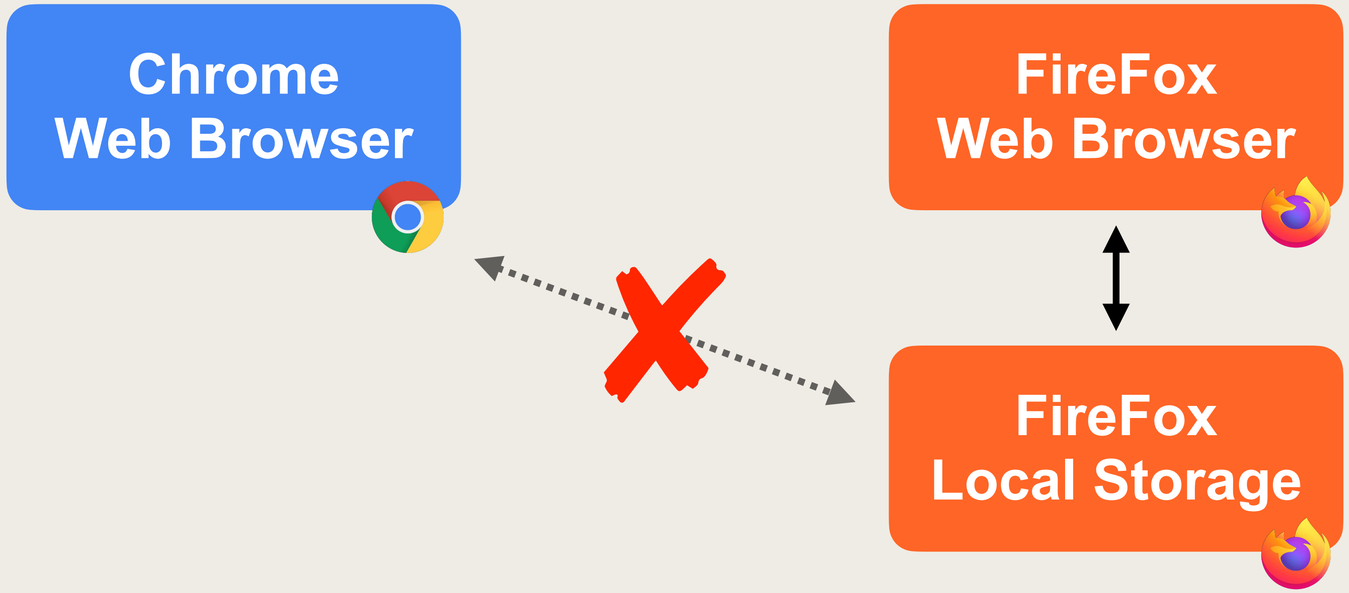
- Same origin `http://localhost:4200 == http://localhost:4200`

- Different origin `http://localhost:4200 != http://localhost:8080`

Different ports

Local Storage

- For Local Storage ... data is NOT shared between different web browsers
- For example ... Chrome can not access Local Storage of FireFox etc ...



Factors to consider

- Data in web storage is stored as plain text ... NOT encrypted
  - Do not use it to store sensitive info such as credit card numbers etc ...
  - Be aware that power users may tinker with files on their computer

- Your app should be resilient to still work if storage is not available
  - The user may clear their browser cache etc ...
  - Your app should use reasonable defaults

Web Storage API

- The API works the same for session storage or local storage
  - Based on concept of key and value
  - Key and value are always strings

Only pages from the same origin can access the data

- Remember data is scoped to the page origin: **`protocol + hostname + port`**
  - Store item using: `storage.setItem(key, value)`
  - Retrieve item using: `storage.getItem(key)`

Web Storage API

Only pages from the same origin can access the data

Method	Description
`setItem(key, value) : void`	Set the key with given value If already exists, will update the value
`getItem(key) : string`	Returns the key value If it doesn't exist, return null
`removeItem(key): void`	Remove item key If it doesn't exist, do nothing
`clear(): void`	Empties all keys out of storage

www.luv2code.com/web-storage-api

Development Process - Angular

1. Update CartService to read data from session storage

2. Add new method in CartService: persistCartItems()

3. Modify computeCartTotals() to call new method: persistCartItems()

Step 1: Update CartService to read data from session storage

File: cart.service.ts

```
export class CartService {

  cartItems: CartItem[] = [];
  ...

  storage : Storage = sessionStorage;

  constructor() {

    // read data from storage
    let data = JSON.parse(this.storage.getItem('cartItems'));

    if (data != null) {
      this.cartItems = data;

      // compute totals based on the data that is read from storage
      this.computeCartTotals();
    }

  }
```

Reference to web browser's session storage

Key

You can use any name just stay consistent

`JSON.parse(...)`
Reads JSON string and converts to object

Step 2: Add new method in CartService: persistCartItems()

File: cart.service.ts

```
export class CartService {

  cartItems: CartItem[] = [];
  ...

  storage : Storage = sessionStorage;

  persistCartItems() {
    this.storage.setItem('cartItems', JSON.stringify(this.cartItems));
  }
```

JSON.stringify(...)
Converts object to JSON string

Key

Value

Step 3: Modify computeCartTotals() to call new method: persistCartItems()

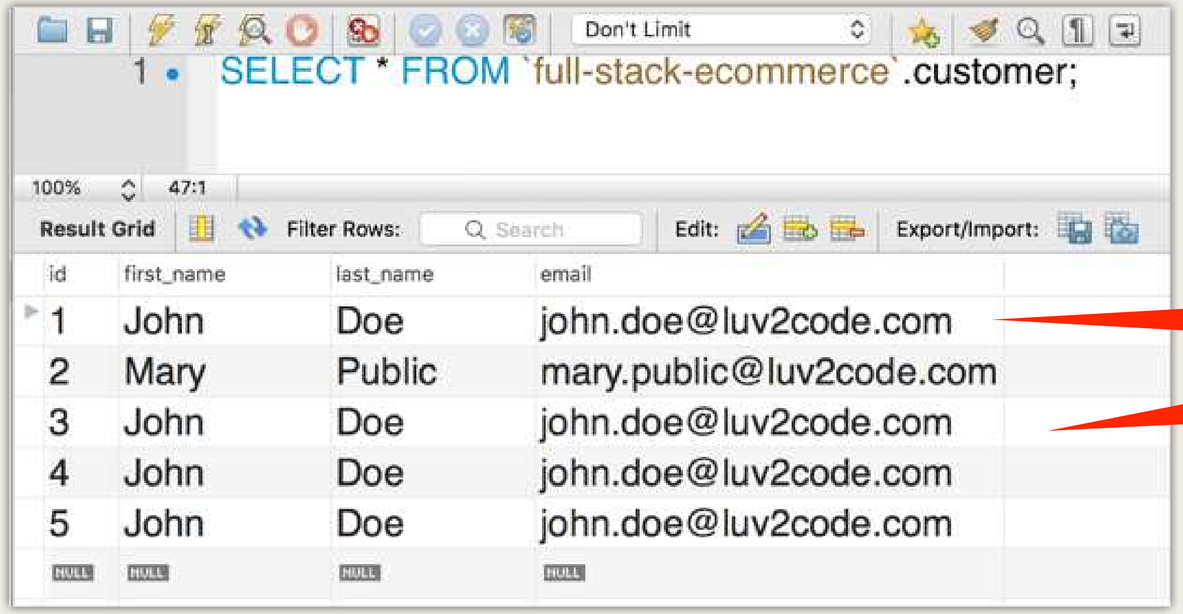
File: cart.service.ts

```
export class CartService {

  cartItems: CartItem[] = [];
  ...

  storage : Storage = sessionStorage;

  persistCartItems() {
    this.storage.setItem('cartItems', JSON.stringify(this.cartItems));
  }

  computeCartTotals() {
    ...
    ...

    // persist cart data
    this.persistCartItems();
  }
```

Handle Customers by Email

Code Refactoring

- Currently, if we perform multiple checkouts with **same email address**
- We have multiple customer entries in customer table



Our Solution

- A single customer is associated with multiple orders
- On the backend, in our CheckoutServiceImpl
  - Check database if customer already exists based on email
  - If so then use the existing customer from database
  - Else we have a new customer

Development Process

1. Remove existing data from database tables

2. Modify database schema to only allow unique email addresses

3. Add new method to CustomerRepository
    1. findByEmail(...)

4. Update CheckoutServiceImpl
    1. Check if customer already exists ... if so then use the existing customer

Step 1: Remove existing data from database tables

Run this SQL in MySQL Workbench

```
USE `full-stack-ecommerce`;

-- clean up previous database tables

SET FOREIGN_KEY_CHECKS=0;

TRUNCATE customer;
TRUNCATE orders;
TRUNCATE order_item;
TRUNCATE address;

SET FOREIGN_KEY_CHECKS=1;
```

the `truncate` command removes ALL rows from a table

In general
- `truncate` is faster than `delete`
- `truncate` resets auto_increment back to starting value

Step 2: Modify database schema to only allow unique email addresses

Run this SQL in MySQL Workbench

```
USE `full-stack-ecommerce`;

-- clean up previous database tables

SET FOREIGN_KEY_CHECKS=0;

TRUNCATE customer;
TRUNCATE orders;
TRUNCATE order_item;
TRUNCATE address;

SET FOREIGN_KEY_CHECKS=1;


-- make the email address unique

ALTER TABLE customer ADD UNIQUE (email);
```

Database constraint

MySQL will throw an error if you attempt to insert duplicate email

Step 3: Add new method to CustomerRepository

- Spring Data REST and Spring Data JPA supports "query methods"
- Spring will construct a query based on method naming conventions

File: CustomerRepository.java

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    Customer findByEmail(String theEmail);

}
```

Magic!

Step 3: Add new method to CustomerRepository

- Find a customer based on email

File: CustomerRepository.java

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    Customer findByEmail(String theEmail);

}
```

Method returns null if not found

Behind the scenes, Spring will execute a query similar to this

```
SELECT * FROM Customer c WHERE c.email = theEmail
```

Step 4: Update CheckoutServiceImpl

File: CheckoutServiceImpl.java

```
public PurchaseResponse placeOrder(Purchase purchase) {

    // retrieve the order info from dto
    Order order = purchase.getOrder();
    ...
    ...
    // populate customer with order
    Customer customer = purchase.getCustomer();

    // check if this is an existing customer ...
    String theEmail = customer.getEmail();

    Customer customerFromDB = customerRepository.findByEmail(theEmail);

    if (customerFromDB != null) {
        // we found them ... let's assign them accordingly
        customer = customerFromDB;
    }

    // add the order to the customer
    customer.add(order);

    // save to the database
    customerRepository.save(customer);

    // return a response
    return new PurchaseResponse(orderTrackingNumber);
}
```

Method returns null if not found

luv2code

Refactoring Backend Configs

Development Process

1. Fix deprecated method for Spring Data REST
2. Configure CORS mapping for Spring Data REST
3. Configure CORS mapping for @RestController
4. Disable HTTP PATCH method
5. Modify Spring Data REST Detection Strategy

Step 1: Fix deprecated method for Spring Data REST

File: MyDataRestConfig.java

```
public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config) {

  ...

}
```

Before - deprecated

File: MyDataRestConfig.java

```
public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config, CorsRegistry cors) {

  ...

}
```

After - current

Step 2: Configure CORS mapping for Spring Data REST

File: MyDataRestConfig.java

```
public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config, CorsRegistry cors) {

  ...

  // configure cors mapping
  cors.addMapping("/api/**").allowedOrigins("http://localhost:4200");

}
```

File: application.properties

```
...
spring.data.rest.base-path=/api
```

Step 2: Configure CORS mapping for Spring Data REST

File: MyDataRestConfig.java

```
public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config, CorsRegistry cors) {

  ...

  // configure cors mapping
  cors.addMapping("/api/**").allowedOrigins("http://localhost:4200");

}
```

Now we can remove the @CrossOrigin from JpaRepositories

File: StateRepository.java

```
@CrossOrigin("http://localhost:4200")
@RepositoryRestResource
public interface StateRepository extends JpaRepository<State, Integer> {

    ... ...

}
```

Step 2: Configure CORS mapping for Spring Data REST

Move configuration for "allowed origins" to application.properties

File: MyDataRestConfig.java

```
@Configuration
public class MyDataRestConfig implements RepositoryRestConfigurer {


    @Value("${allowed.origins}")
    private String[] theAllowedOrigins;


    @Override
    public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config, CorsRegistry cors) {

      ...
      // configure cors mapping
      cors.addMapping("/api/**").allowedOrigins(theAllowedOrigins);

    }
}
```

1

2

File: application.properties

```
allowed.origins=http://localhost:4200
...
```

If you have multiple origins, then give a comma-delimited list

Step 2: Configure CORS mapping for Spring Data REST

File: MyDataRestConfig.java

```
@Configuration
public class MyDataRestConfig implements RepositoryRestConfigurer {

    @Value("${allowed.origins}")
    private String[] theAllowedOrigins;

    @Override
    public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config, CorsRegistry cors) {

      ...
      // configure cors mapping
      cors.addMapping(config.getBasePath() + "/**").allowedOrigins(theAllowedOrigins);

    }
}
```

Use existing property:
`spring.data.rest.base-path`

Available in the `config` object

File: application.properties

```
spring.data.rest.base-path=/api
...
```

Step 3: Configure CORS mapping for @RestController

@RestController configuration is separate from Spring Data REST configuration

File: MyA

```
@Con
public class MyAppConfig implements WebMvcConfigurer {

    @Value("${spring.data.rest.base-path}")
    private String basePath;

    @Value("${allowed.origins}")
    private String[] theAllowedOrigins;

    @Override
    public void addCorsMappings(CorsRegistry cors) {

        // configure cors mapping
        cors.addMapping(basePath + "/**").allowedOrigins(theAllowedOrigins);

    }
}
```

File: application.properties

```
spring.data.rest.base-path=/api
allowed.origins=http://localhost:4200
...
```

Step 3: Configure CORS mapping for @RestController

Now we can remove the @CrossOrigin from @RestController

File: CheckoutController.java

```
@CrossOrigin("http://localhost:4200")
@RestController
@RequestMapping("/api/checkout")
public class CheckoutController {

   ...

}
```

Step 4: Disable HTTP PATCH method

- Previously, we configured the Spring Data REST APIs for read-only
  - Disabled: HTTP GET, POST and DELETE

- However, we need to ALSO disable: HTTP PATCH

Step 4: Disable HTTP PATCH method

File: MyDataRestConfig.java

```
@Override
public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config, CorsRegistry cors) {

    HttpMethod[] theUnsupportedActions = {HttpMethod.PUT, HttpMethod.POST, HttpMethod.DELETE, HttpMethod.PATCH};

    // disable HTTP methods for ProductCategory: PUT, POST, DELETE and PATCH
    disableHttpMethods(Product.class, config, theUnsupportedActions);
    disableHttpMethods(ProductCategory.class, config, theUnsupportedActions);
    disableHttpMethods(Country.class, config, theUnsupportedActions);
    disableHttpMethods(State.class, config, theUnsupportedActions);
    ...
}
```

Step 5: Modify Spring Data REST Detection Strategy

- By default, Spring Data REST will expose REST APIs for Spring Data Repositories
- This works in general ... but you may not want to expose all
- For example, in our case, we have CustomerRepository
  - BUT ... we don't want to expose this as a REST API
  - We only want to use it internally .... like for checking if customer email exists
- REST endpoint `/api/customers` is currently exposed .... let's fix this ASAP!

Step 5: Modify Spring Data REST Detection Strategy

- Spring Data REST has different detection strategies

Strategy	Description
`ALL`	**Exposes all Spring Data repositories regardless of their Java visibility or annotation configuration**
`DEFAULT`	**Exposes public Spring Data repositories or ones explicitly annotated with `@RepositoryRestResource` and its exported attribute not set to false**
`VISIBILITY`	**Exposes only public Spring Data repositories regardless of annotation configuration**
`ANNOTATED`	**Only exposes Spring Data repositories explicitly annotated with `@RepositoryRestResource` and its exported attribute not set to false**

Good approach for our app

Step 5: Modify Spring Data REST Detection Strategy

ANNOTATED	Only exposes Spring Data repositories explicitly annotated with `@RepositoryRestResource` and its exported attribute not set to false

File: StateRepository.java

```
@RepositoryRestResource(exported = true)
public interface StateRepository extends JpaRepository<State, Integer> {

    ... ...

}
```

Default value of exported is `true` ... so we don't have to list it

File: StateRepository.java

```
@RepositoryRestResource
public interface StateRepository extends JpaRepository<State, Integer> {

    ... ...

}
```

Step 5: Modify Spring Data REST Detection Strategy

- Configure detection strategy

ANNOTATED	Only exposes Spring Data repositories explicitly annotated with `@RepositoryRestResource` and its exported attribute not set to false

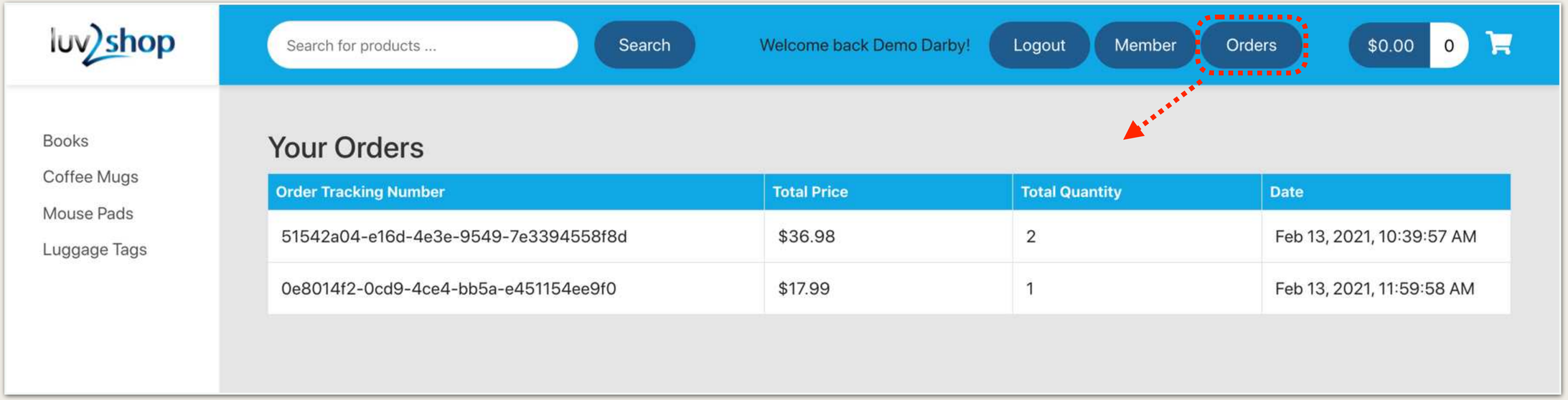
File: application.properties

```
spring.data.rest.detection-strategy=ANNOTATED
... ...
```

View Order History

View Order History

- Allow users to view their previous order history
- This feature is only available to logged in / authenticated users



Front End and Back End

- Split up the development across multiple videos
- We'll start working on Spring Boot back end
- Then we'll work on Angular front end

Development Process - Spring Boot

1. Create OrderRepository REST API
2. Update configs to make OrderRepository REST API read only

Step 1: Create OrderRepository REST API

- Expose endpoint to search for orders by the customer's email address

HTTP Method	Endpoint
`GET`	`/api/orders/search/findByCustomerEmail?email=demo@luv2code.com`

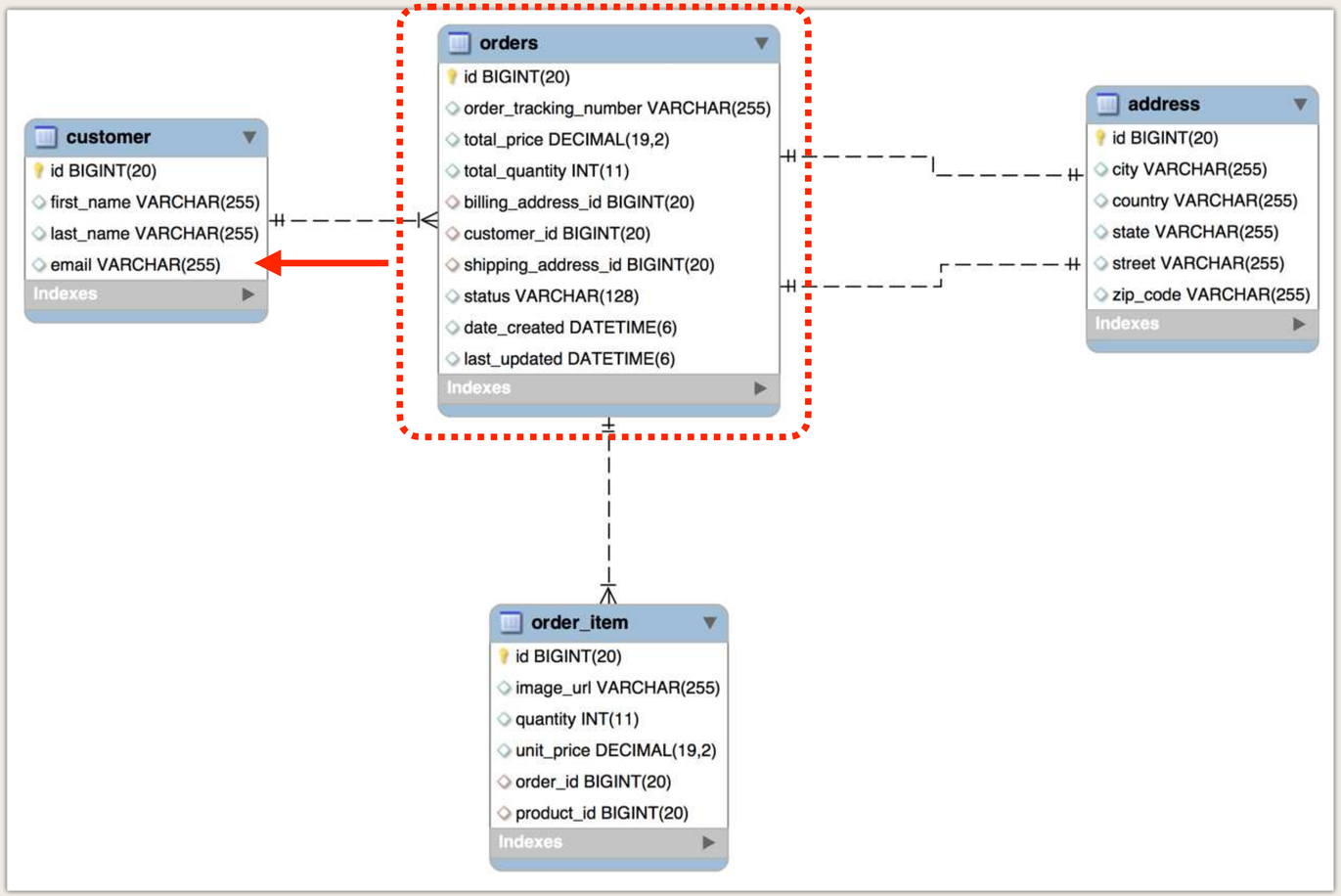
Step 1: Create OrderRepository REST API

- Spring Data REST and Spring Data JPA supports "query methods"
- Spring will construct a query based on method naming conventions

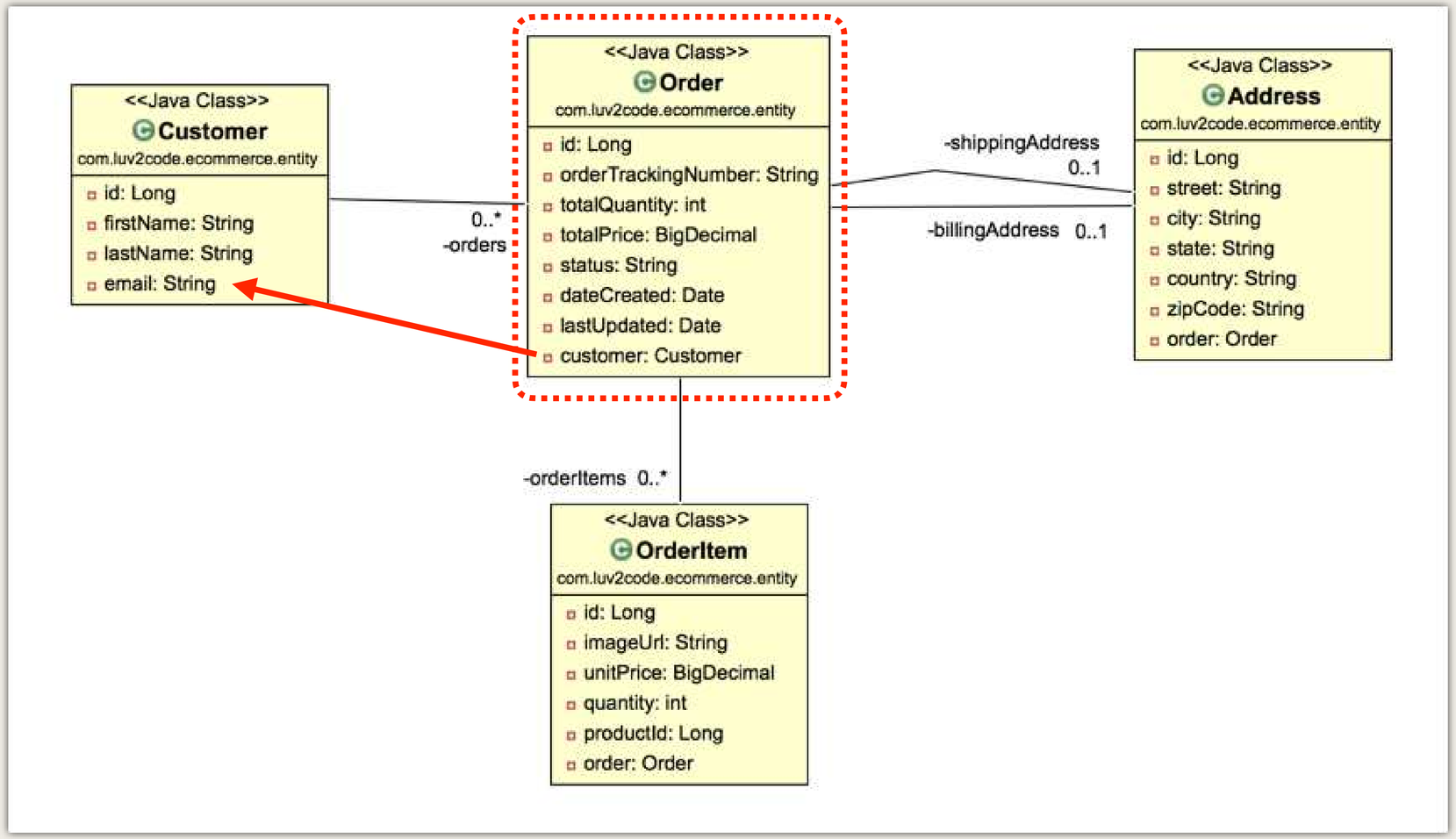
File: OrderRepository.java

```
@RepositoryRestResource
public interface OrderRepository extends JpaRepository<Order, Long> {

    Page<Order> findByCustomerEmail(@Param("email") String email, Pageable pageable);

}
```

Database Diagram



Class Diagram - Entity Classes



Step 1: Create OrderRepository REST API

File: OrderRepository.java

```
@RepositoryRestResource
public interface OrderRepository extends JpaRepository<Order, Long> {

    Page<Order> findByCustomerEmail(@Param("email") String email, Pageable pageable);

}
```

Behind the scenes, Spring will execute a query similar to this

```
SELECT * FROM orders
LEFT OUTER JOIN customer
ON orders.customer_id=customer.id
WHERE customer.email=:email
```

Step 1: Create OrderRepository REST API

File: OrderRepository.java

```
@RepositoryRestResource
public interface OrderRepository extends JpaRepository<Order, Long> {

    Page<Order> findByCustomerEmail(@Param("email") String email, Pageable pageable);

}
```

http://localhost:8080/api/orders/search/findByCustomerEmail?email=demo@luv2code.com

To pass data to REST API

Step 2: Update configs to make OrderRepository REST API read only

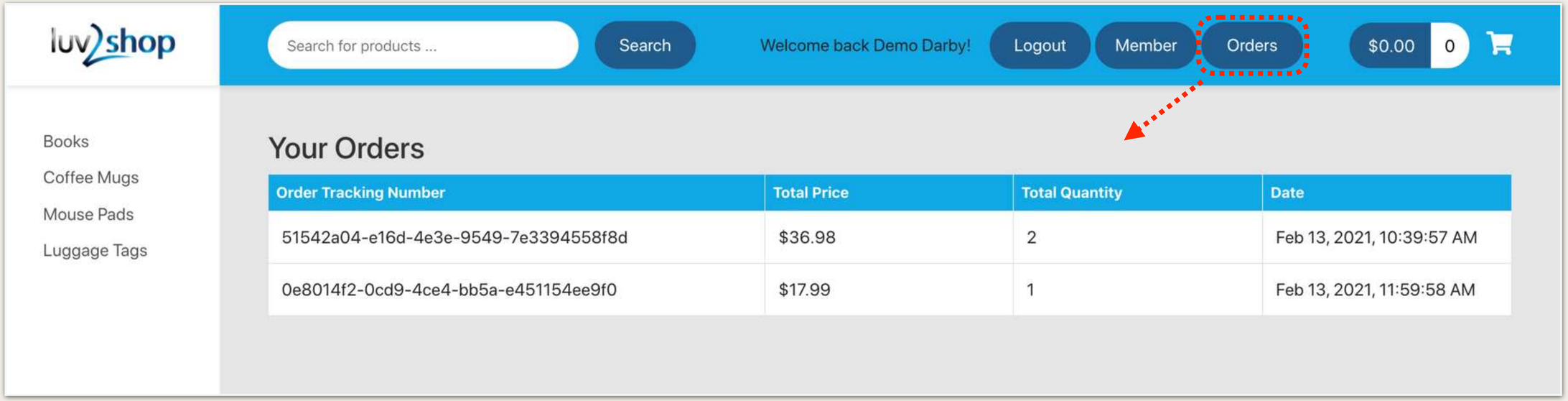
File: MyDataRestConfig.java

```
@Configuration
public class MyDataRestConfig implements RepositoryRestConfigurer {

 @Override
 public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config, CorsRegistry cors) {

    HttpMethod[] theUnsupportedActions = {HttpMethod.PUT, HttpMethod.POST,
                                          HttpMethod.DELETE, HttpMethod.PATCH};

     // disable HTTP methods for ProductCategory: PUT, POST and DELETE
    disableHttpMethods(Product.class, config, theUnsupportedActions);
    disableHttpMethods(ProductCategory.class, config, theUnsupportedActions);
    ...
    disableHttpMethods(Order.class, config, theUnsupportedActions);

  }

  ...
}
```

View Order History

View Order History - Front End

- We have the Spring Boot back end developed
- Now let's focus on the Angular front end



Development Process - Angular

1. Keep track of logged in user's email with web browser storage
2. Create OrderHistory class
3. Develop OrderHistory service
4. Generate order-history component
5. Update template text in HTML page
6. Add "Orders" button to login-status component
7. Define protected route for order-history component

Step 1: Keep track of logged in user's email with web browser storage

- We know that we will need to retrieve orders by user's email address
- Once they log in, let's keep track of the user's email address
  - Store the email address in the web browser storage
  - Other Angular components can retrieve it at a later time
- Use Browser **session** storage
  - Data is only available during the current web browser session

Step 1: Keep track of logged in user's email with web browser storage

File: login-status.component.ts

```
export class LoginStatusComponent implements OnInit {
  ...
  storage: Storage = sessionStorage;
  ...

  getUserDetails() {
    if (this.isAuthenticated) {

      // Fetch the logged in user details (user's claims)
     this.oktaAuthService.getUser().then(
        (res) => {
          this.userFullName = res.name;

          // retrieve the user's email from authentication response
          const theEmail = res.email;

          // now store the email in browser storage
          this.storage.setItem('userEmail', JSON.stringify(theEmail));

        }
      );
    }
  }
}
```

Reference to web browser's session storage

Store user's email in web browser's session storage

Key

Value

Step 2: Create OrderHistory class

```
$ ng generate class common/OrderHistory
```

File: order-history.ts

```
export class OrderHistory {
    id: string;
    orderTrackingNumber: string;
    totalPrice: number;
    totalQuantity: number;
    dateCreated: Date;
}
```

Step 3: Develop OrderHistory service

```
$ ng generate service services/OrderHistory
```

Step 3: Develop OrderHistory service

File: order-history.service.ts

```
export class OrderHistoryService {

  private orderUrl = 'http://localhost:8080/api/orders';

  constructor(private httpClient: HttpClient) { }

  getOrderHistory(theEmail: string): Observable<GetResponseOrderHistory> {

    // need to build URL based on the customer email
    const orderHistoryUrl = `${this.orderUrl}/search/findByCustomerEmail?email=${theEmail}`;

    return this.httpClient.get<GetResponseOrderHistory>(orderHistoryUrl);
  }
}

interface GetResponseOrderHistory {
  _embedded: {
    orders: OrderHistory[];
  }
}
```

Call REST API

Unwraps the JSON from Spring Data REST _embedded entry

Step 4: Generate order-history component

```
$ ng generate component components/OrderHistory
```

Step 4: Generate order-history component

File:order-history.component.ts

```
export class OrderHistoryComponent implements OnInit {

  orderHistoryList: OrderHistory[] = [];
  storage: Storage = sessionStorage;

  constructor(private orderHistoryService: OrderHistoryService) { }

  ngOnInit(): void {
    this.handleOrderHistory();
  }

  handleOrderHistory() {

    // read the user's email address from browser storage
    const theEmail = JSON.parse(this.storage.getItem('userEmail'));

    // retrieve data from the service
    this.orderHistoryService.getOrderHistory(theEmail).subscribe(
      data => {
        this.orderHistoryList = data._embedded.orders;
      }
    );
  }
}
```

Reference to web browser's session storage

Inject OrderHistoryService

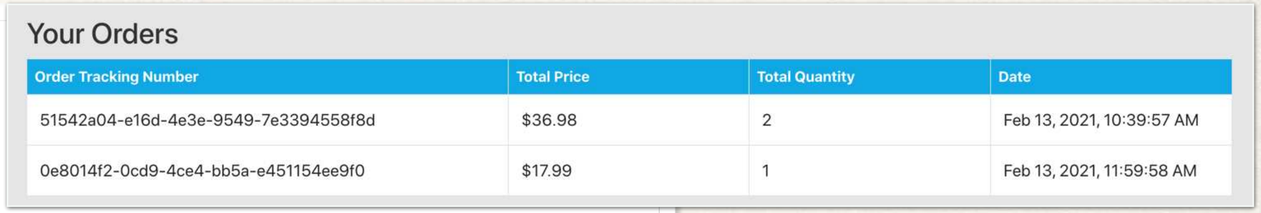
Key

assign data

Step 5: Update template text in HTML page

File: order-history.component.html

```
<h3>Your Orders</h3>

<div *ngIf="orderHistoryList.length > 0">
    <table class="table table-bordered">
        <tr>
            <th width="20%">Order Tracking Number</th>
            <th width="10%">Total Price</th>
            <th width="10%">Total Quantity</th>
            <th width="10%">Date</th>
        </tr>

        <tr *ngFor="let tempOrderHistory of orderHistoryList">
            <td>
                {{ tempOrderHistory.orderTrackingNumber }}
            </td>
            <td>
                {{ tempOrderHistory.totalPrice | currency: 'USD' }}
            </td>
            <td>
                {{ tempOrderHistory.totalQuantity }}
            </td>
            <td>
                {{ tempOrderHistory.dateCreated | date: 'medium' }}
            </td>
        </tr>
    </table>
</div>
```



Your Orders

Order Tracking Number	Total Price	Total Quantity	Date
51542a04-e16d-4e3e-9549-7e3394558f8d	$36.98	2	Feb 13, 2021, 10:39:57 AM
0e8014f2-0cd9-4ce4-bb5a-e451154ee9f0	$17.99	1	Feb 13, 2021, 11:59:58 AM

Loop over orderHistoryList

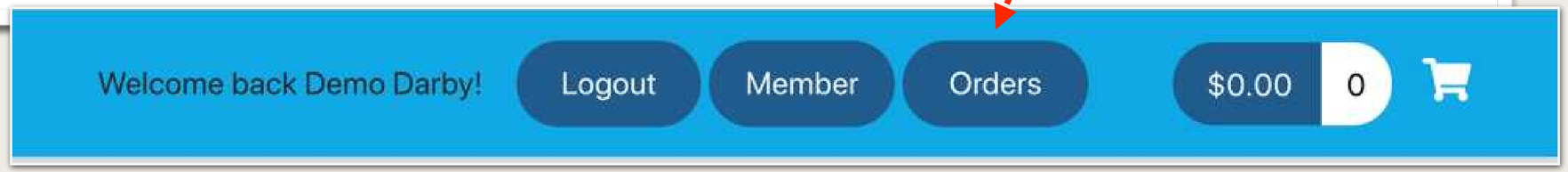
Other date formats: short, medium, long, full ...

https://angular.io/api/common/DatePipe

Step 6: Add "Orders" button to login-status component

File: login-status.component.html

```
<div class="login-status-header">

    <div *ngIf="isAuthenticated && userFullName" class="login-status-user-info">
        Welcome back {{ userFullName }}!
    </div>
    ...

    <button *ngIf="isAuthenticated" routerLink="/members" class="security-btn">Member</button>

    <button *ngIf="isAuthenticated" routerLink="/order-history" class="security-btn">Orders</button>

</div>
```



Step 7: Define protected route for order-history component

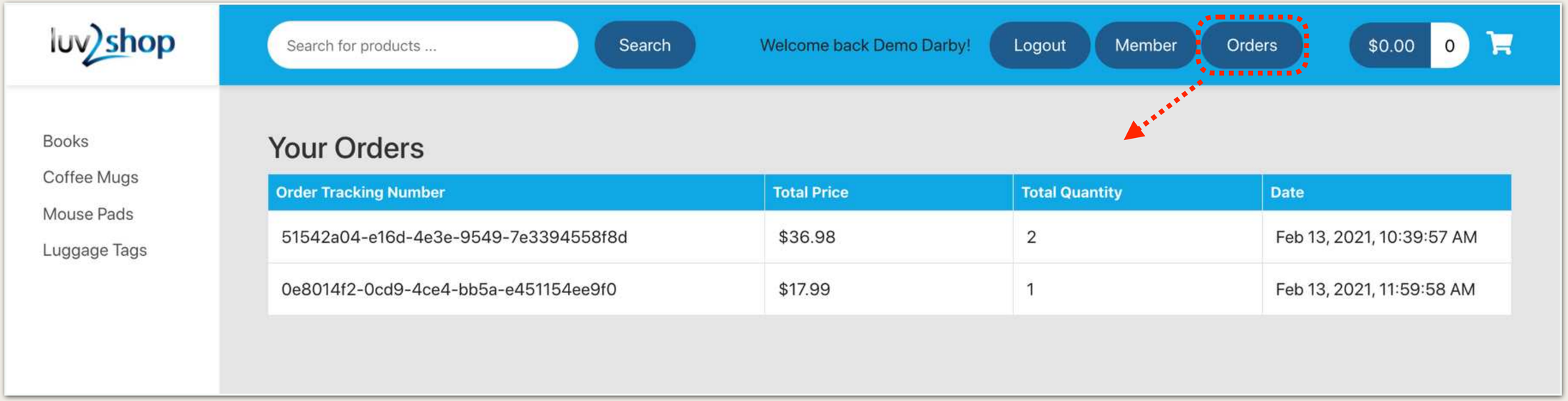
File: app.module.ts

```
const routes: Routes = [

 {path: 'order-history', component: OrderHistoryComponent, canActivate: [ OktaAuthGuard ]},
  ...
];
```

Secure Order History Backend

View Order History

- The Order History functionality is working
- However, the REST API for `/api/orders` is unsecured



Secure Order History Backend

- Let's secure the backend
- `/api/orders` should only available to logged in users

HTTP Method	Endpoint
`GET`	`/api/orders/search/findByCustomerEmail?email=demo@luv2code.com`

Development Process - Spring Boot

1. Add Okta Spring Boot Starter to Maven pom.xml
2. Create an App at the Okta Developer website
3. In Spring Boot app, set application properties
4. Protect endpoints in Spring Security configuration class

Step 1: Add Okta Spring Boot Starter to Maven pom.xml

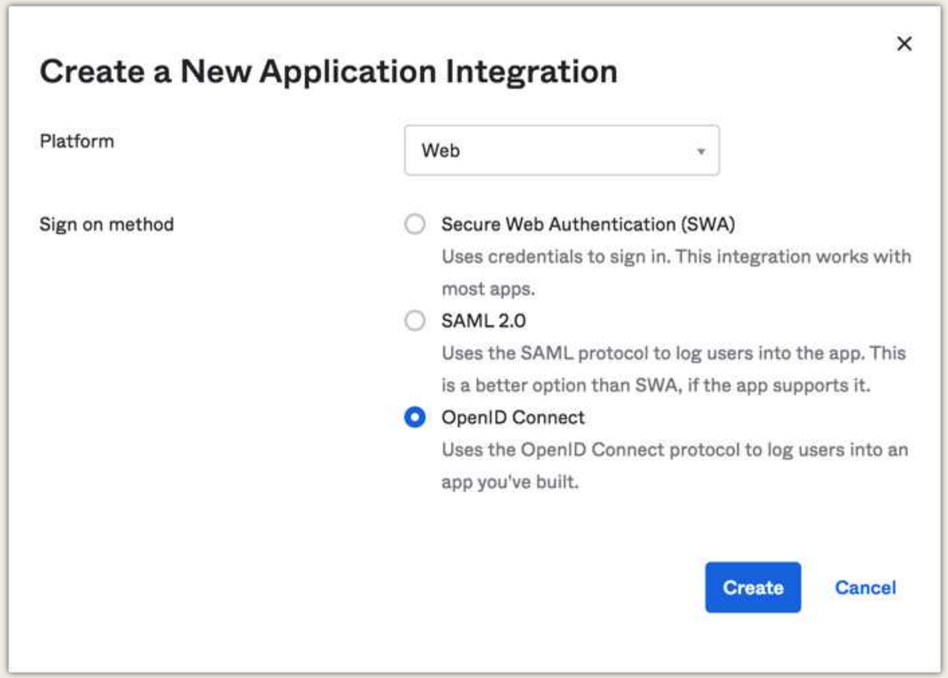
- Okta provides a Spring Boot Starter for OAuth 2 / OpenID Connect
- Simplifies integration and configuration of Spring Security and Okta

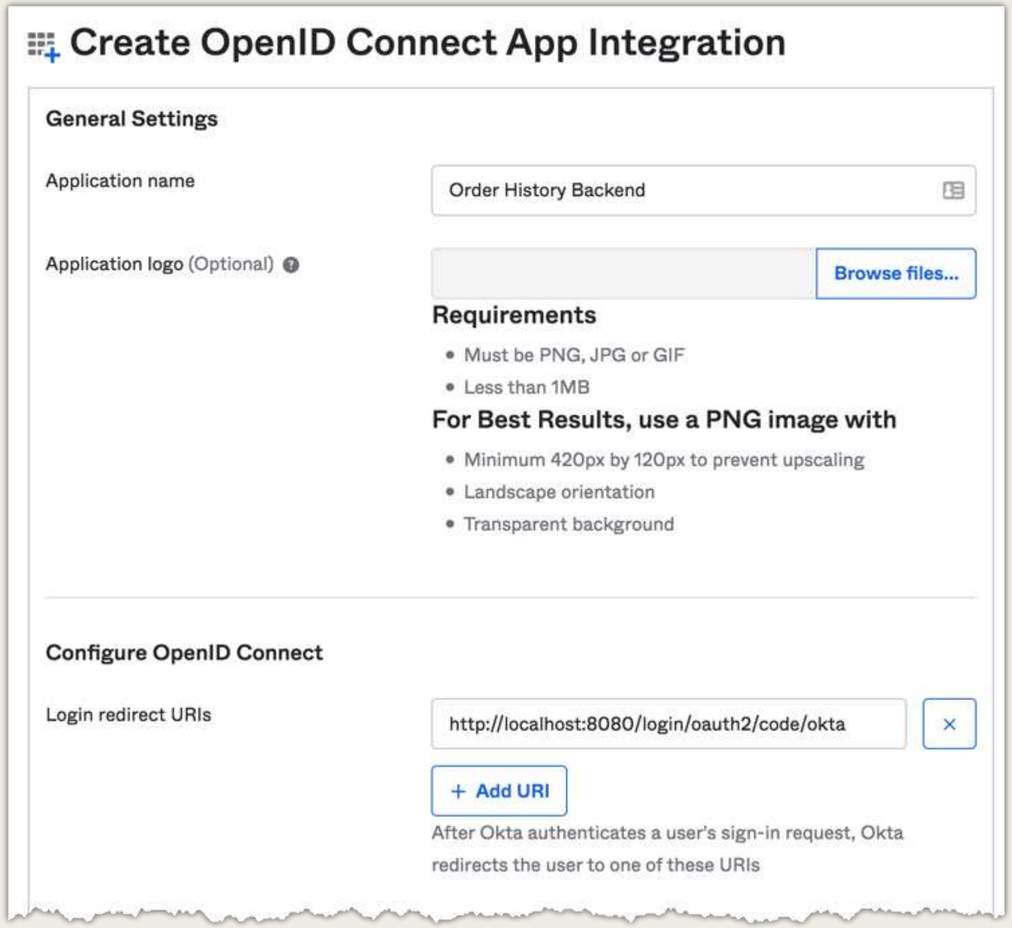
File: pom.xml

```
...
 <dependency>
    <groupId>com.okta.spring</groupId>
    <artifactId>okta-spring-boot-starter</artifactId>
    <version>2.0.1</version>
 </dependency>
...
```

https://github.com/okta/okta-spring-boot

Step 2: Create an App in Okta Developer website

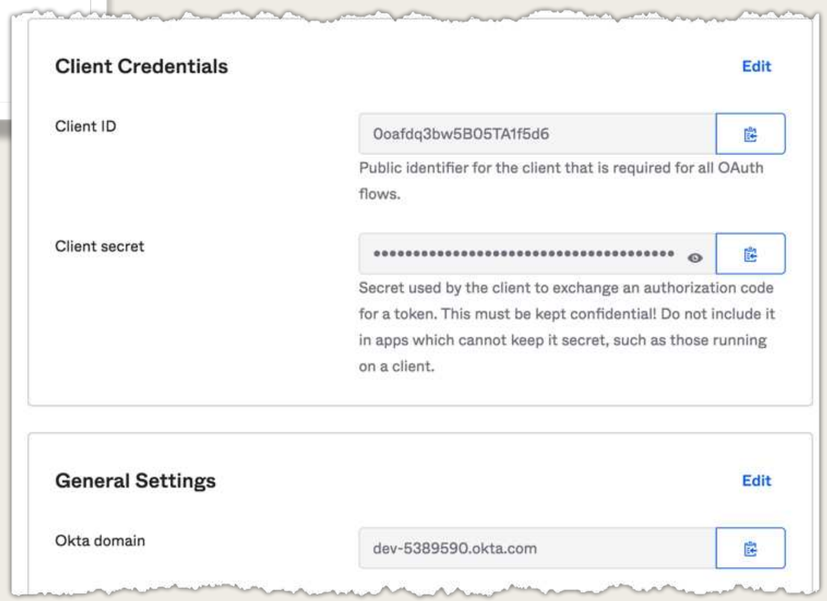




Step 3: In Spring Boot app, set application properties

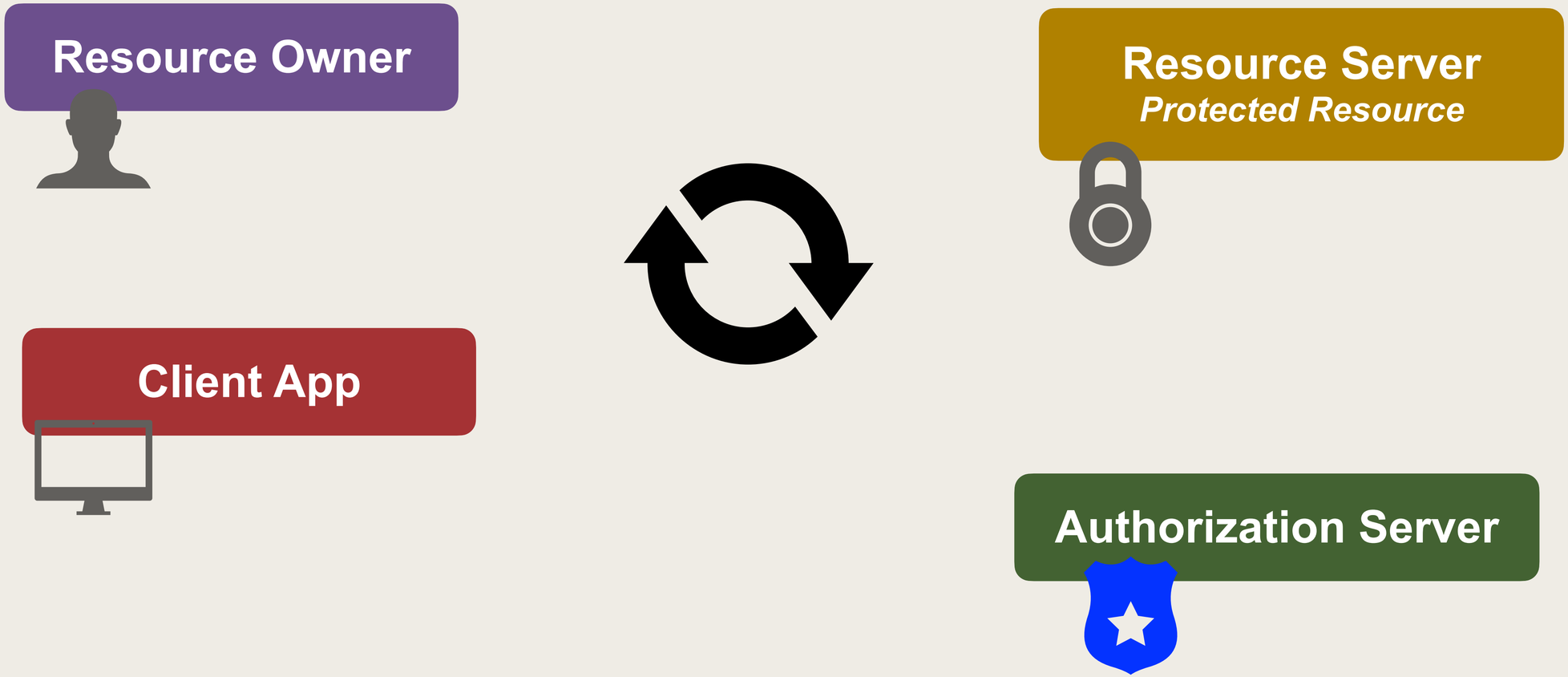
File: application.properties

```
...
okta.oauth2.client-id={yourClientId}
okta.oauth2.client-secret={yourClientSecret}
okta.oauth2.issuer=https://{yourOktaDomain}/oauth2/default
```



Spring Boot application will use these properties to verify / validate JWT access tokens

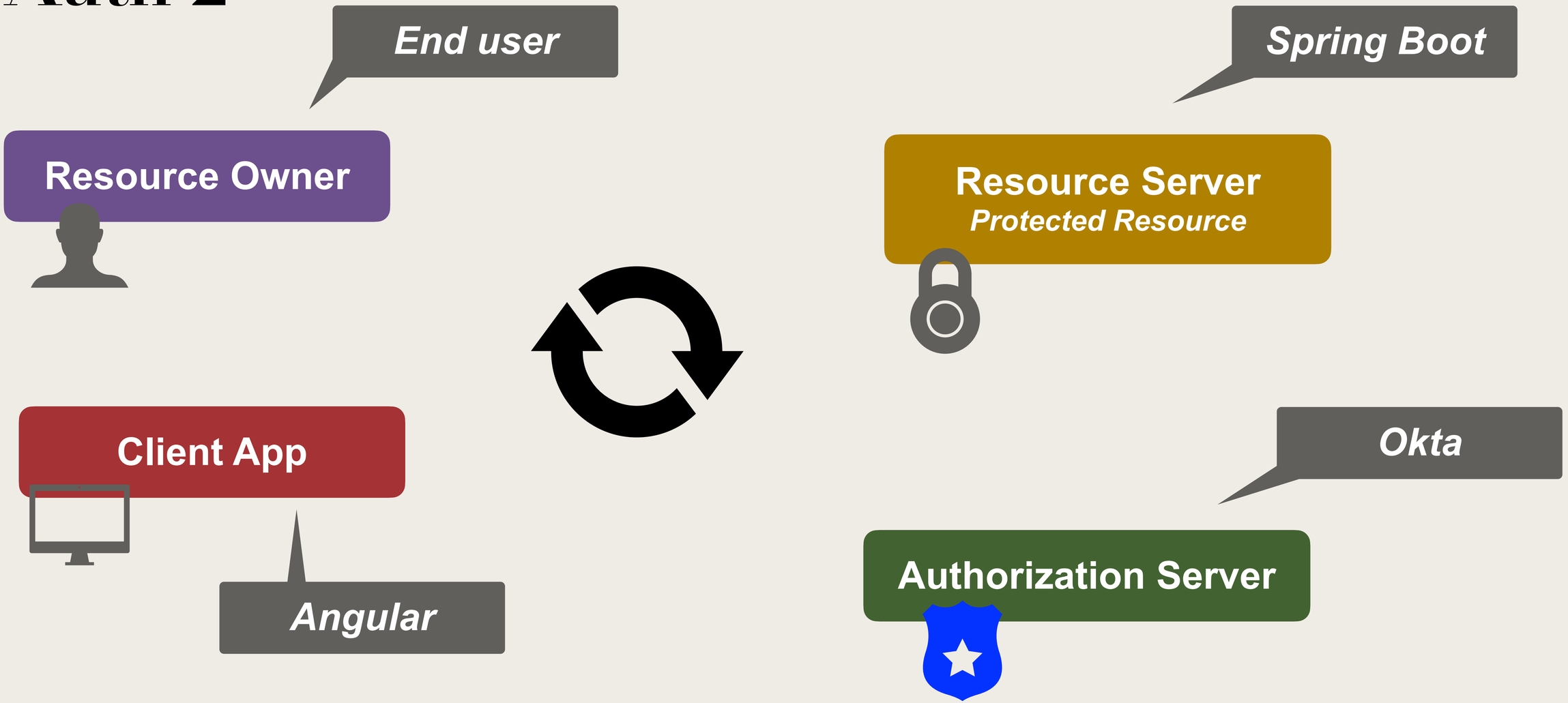
OAuth 2



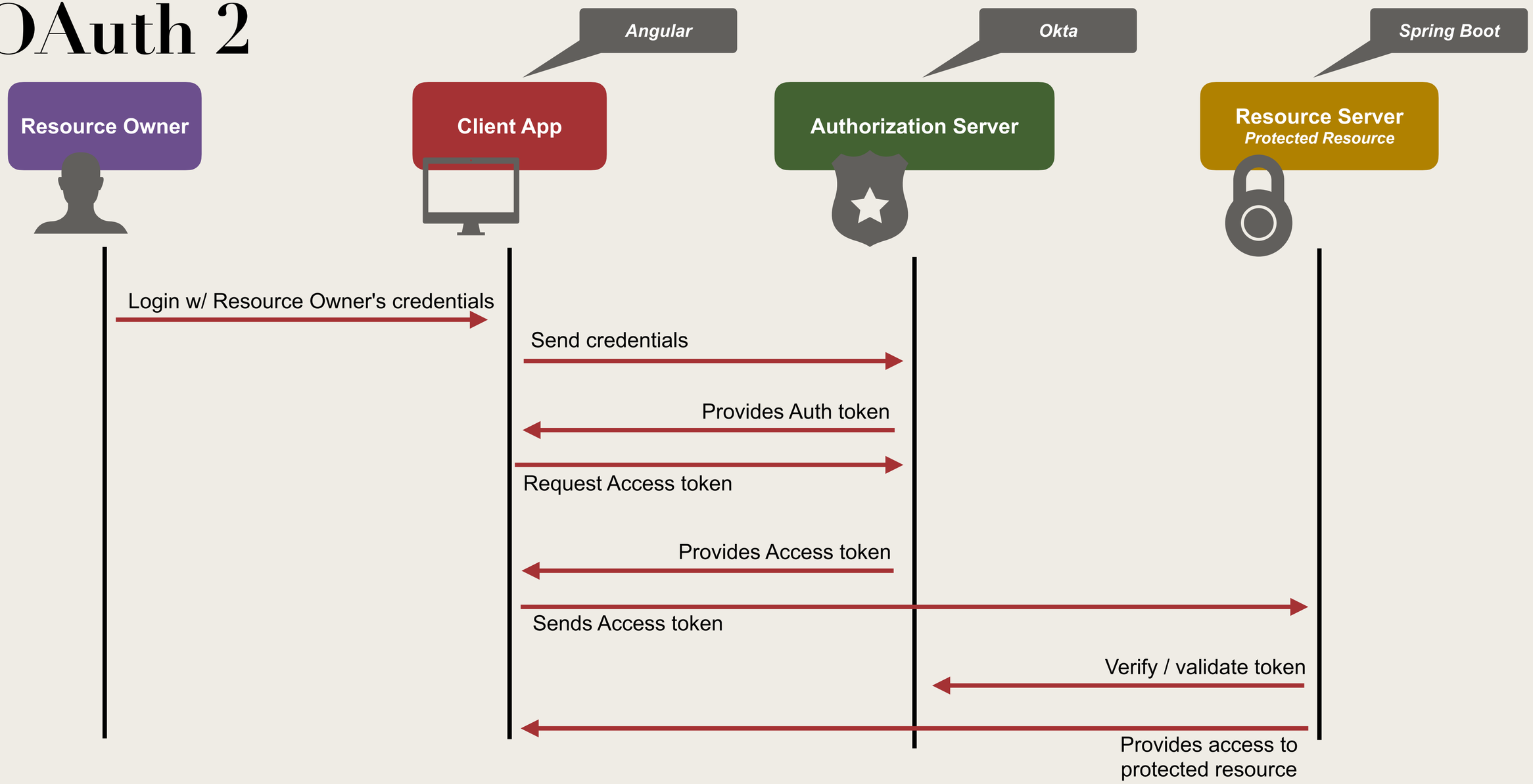
Resource Server

- The "Resource Server" is the app that is hosting our protected resources
  - In our case, it is our Spring Boot app
- The "Resource Server" manages security using access tokens (JWT)
- The access tokens are validated with the "Authorization Server" (Okta)

OAuth 2



OAuth 2



Client sending access token

- Client sends the access token as an HTTP request header

Step 4: Protect endpoints in Spring Security configuration class

File: SecurityConfiguration.java

```
package com.luv2code.ecommerce.config;

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 SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws E

        // protect endpoint /api/orders
        http.authorizeRequests()
                .antMatchers("/api/orders/**")
                .authenticated()
                .and()
                .oauth2ResourceServer()
                .jwt();

        // add CORS filter
        http.cors();

    }
}
```

Protect the endpoint ... only accessible to authenticated users

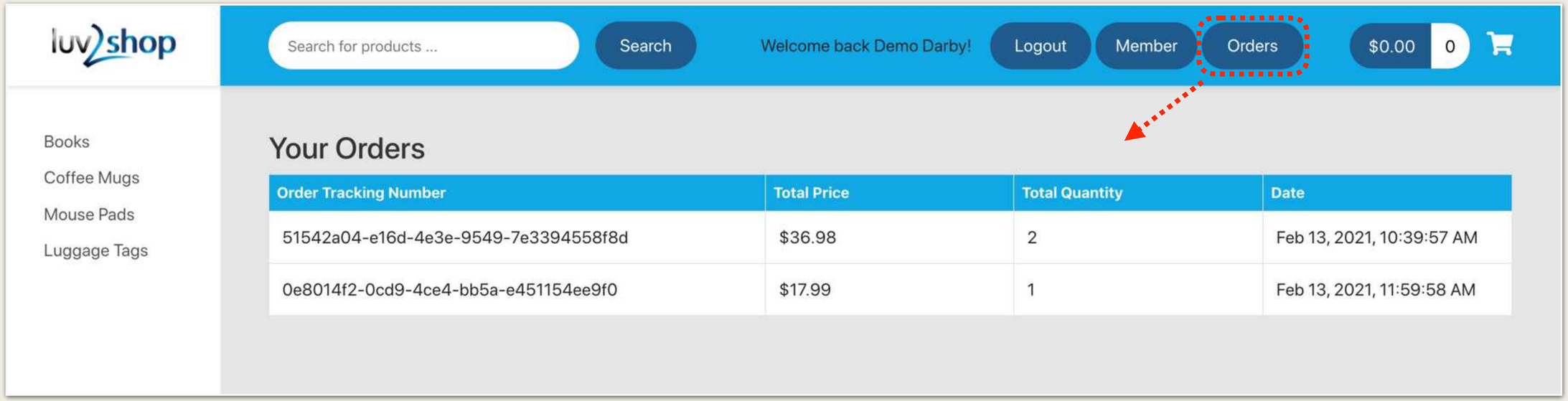
Configures OAuth2 Resource Server support

Enables JWT-encoded bearer token support

Order History Frontend
Pass Access Token

View Order History

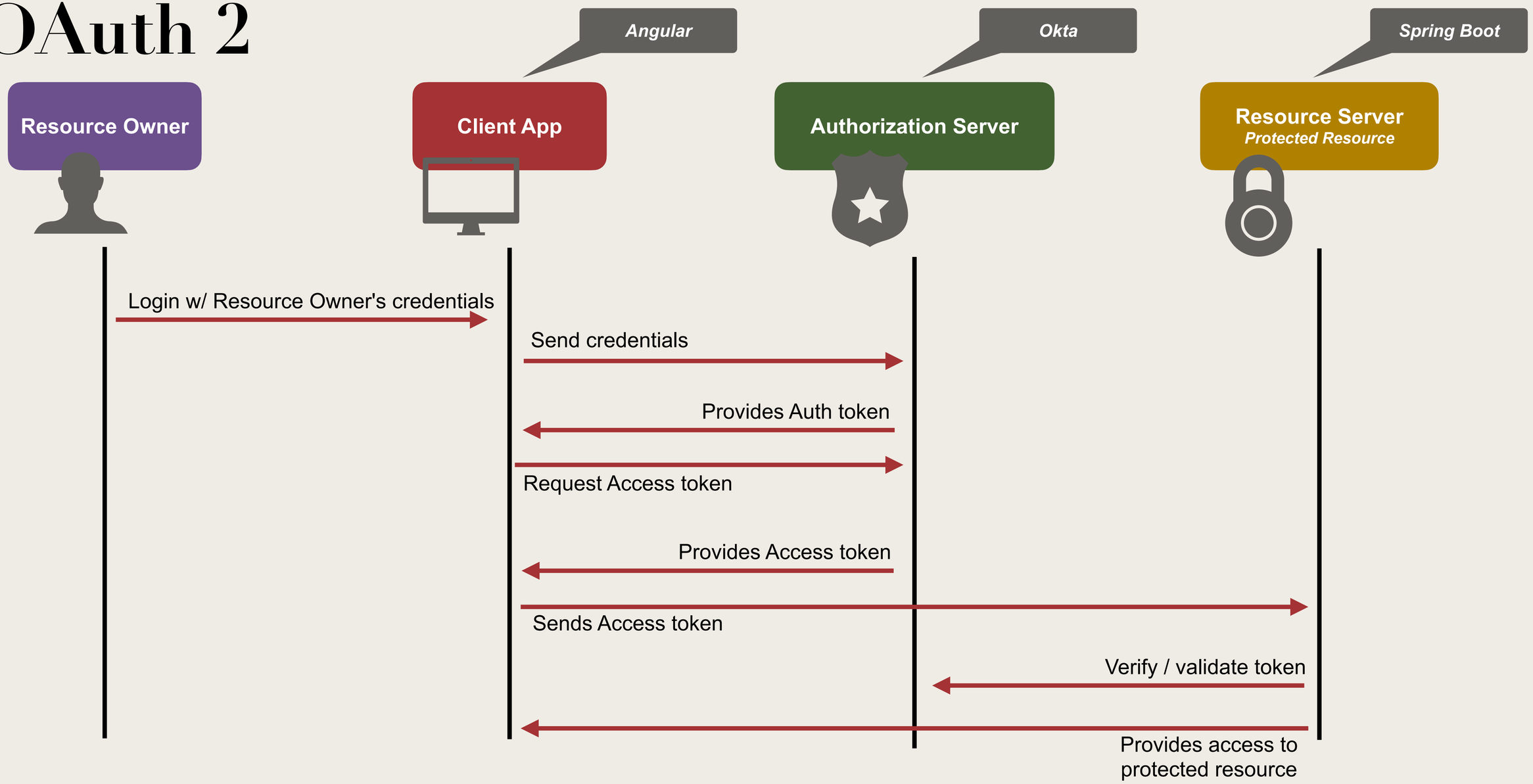
- The REST API for `/api/orders` is secured
- Need to update Angular App to pass the access token



Client sending access token

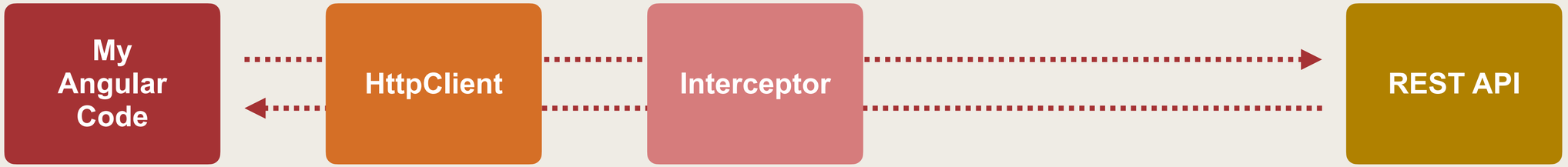
- Client sends the access token as an HTTP request header

OAuth 2



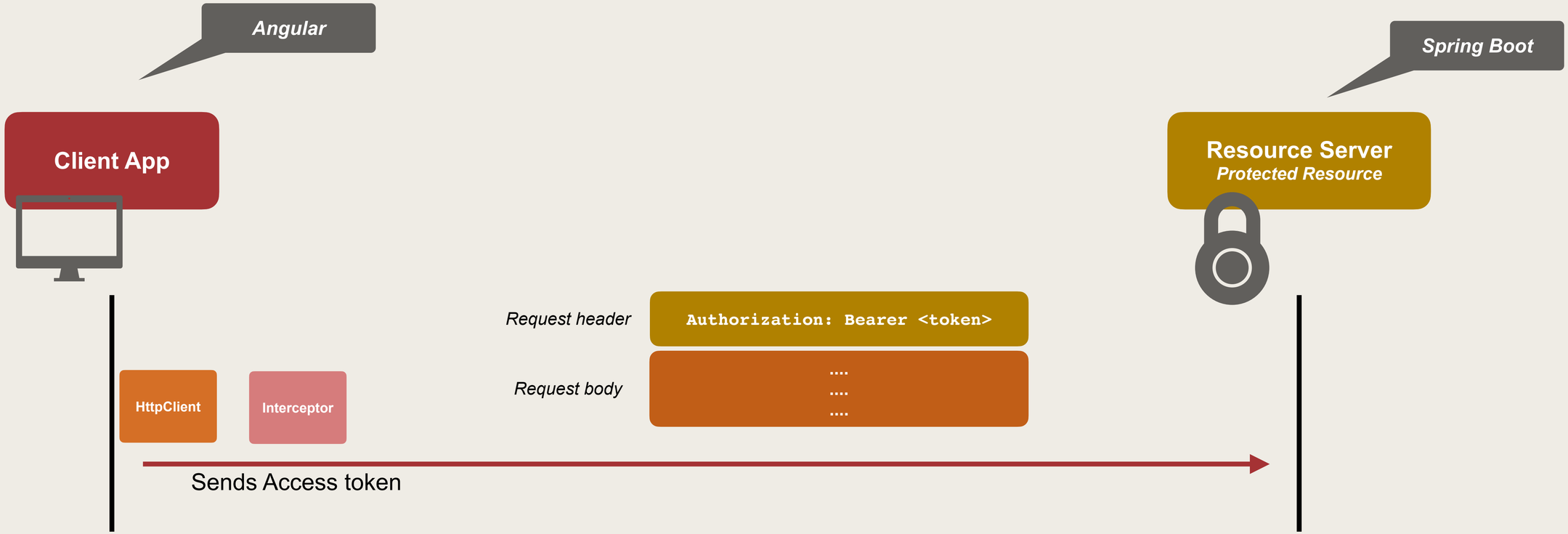
Angular Interceptors

- Angular provides support for interceptors
- An interceptor can intercept HTTP requests / responses
- Useful to perform processing on the HTTP requests / responses



Angular Interceptors

- We will use an interceptor to pass to access token in HTTP request



Development Process

1. Create an interceptor

2. Update app.module.ts to reference interceptor

Step 1: Create an interceptor

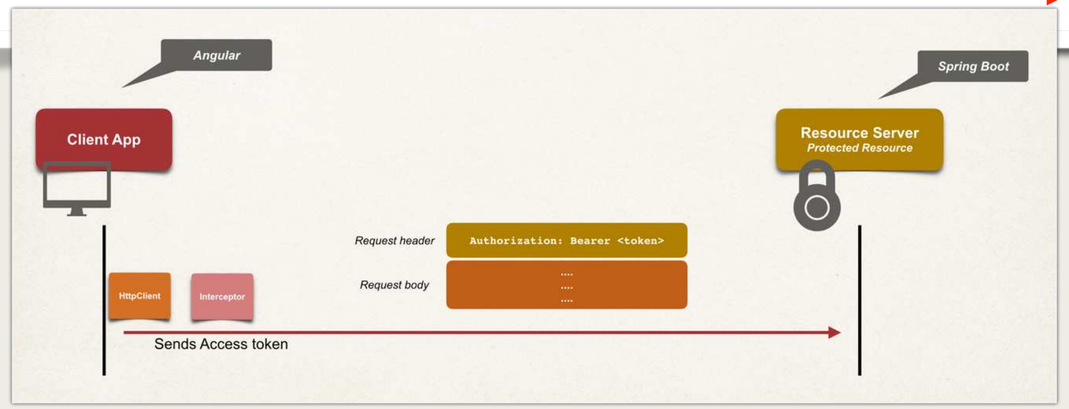
- Develop the interceptor as a service

```
$ ng generate service services/AuthInterceptor
```

Step 1: Create an interceptor

File: auth-interceptor.service.ts

```
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { OktaAuthService } from '@okta/okta-angular';
import { from, Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthInterceptorService implements HttpInterceptor {

  constructor(private oktaAuth: OktaAuthService) { }

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return from(this.handleAccess(request, next));
  }
```



Asynchronous function

Returns a Promise

```
  private async handleAccess(request: HttpRequest<any>, next: HttpHandler): Promise<HttpEvent<any>> {

    // Only add an access token for secured endpoints
    const securedEndpoints = ['http://localhost:8080/api/orders'];

    if (securedEndpoints.some(url => request.urlWithParams.includes(url))) {

      // get access token
      const accessToken = await this.oktaAuth.getAccessToken();

      // clone the request and add new header with access token
      requ
        se
                                                    ' + acce
        }
      });

    }

    return next.handle(request).toPromise();
  }
}
```

Waits for the async call to finish

Async call

Step 2: Update app.module.ts to reference interceptor

File: app.module.ts

```
...

@NgModule({

  declarations: [ ... ]

  imports: [ ... ]

  providers: [ProductService, { provide: OKTA_CONFIG, useValue: oktaConfig },
              {provide: HTTP_INTERCEPTORS, useClass: AuthInterceptorService, multi: true}],

})
...
```

Informs Angular that HTTP_INTERCEPTORS is a token for injecting an array of values

Token for HTTP interceptors

Register our AuthInterceptorService as an HTTP interceptor

Additional Resources

- Angular Interceptors
- Promises
- async / await

www.luv2code.com/angular-additional-resources

Secure Communication with HTTPS

Secure Communication

- We would like to have secure communication for our app
- In the future, we'd like to process credit cards

https://localhost:4200

What is HTTPS?

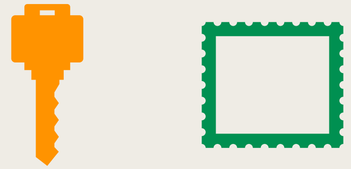
- HTTPS is the Hypertext Transfer Protocol Secure
  - HTTPS is the protocol for encrypting data between web browser and server
  - HTTPS uses the TLS protocol (Transport Layer Security)

- TLS is the successor of SSL. TLS is more secure than SSL.
  - However, in the industry people still use these terms interchangeably
  - TLS / SSL

Using HTTPS

- There are no changes required to the source code
- Configure your server to run using secure keys and certificate
- Then you'll be able access your site with `https://localhost:4200`
- Make note of the use of `https`

Keys and Certs



- To run securely, you will need keys and certificates
  - Provides proof of your server's identity (domain name)
  - Reviewed and signed by a trusted certificate authority (godaddy, verisign, etc)

- In the real-world, you normally have to pay for a certificate
  - For dev/demo purposes, we are going to save money :-)
  - We will create self-signed certificates
  - Browser's will warn about this, but we can safely ignore for dev/demo purposes

Development Process

1. Generate key and self-signed certificate
2. Run Angular App with the key and self-signed certificate
3. Update Spring Boot app with new URL

Step 1: Generate key and self-signed certificate

- Use the free utility: openssl

```
$ openssl req -keyout localhost.key -out localhost.crt ...
```

Name of output keyfile

Name of output certificate

More configs

www.luv2code.com/openssl-setup

Step 2: Run Angular App with the key and self-signed certificate

File: package.json

```
"scripts": {
    "ng": "ng",
    "start": "ng serve --ssl=true --sslKey=./ssl-localhost/localhost.key --sslCert=./ssl-localhost/localhost.crt",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
```

Enable ssl mode

keyfile

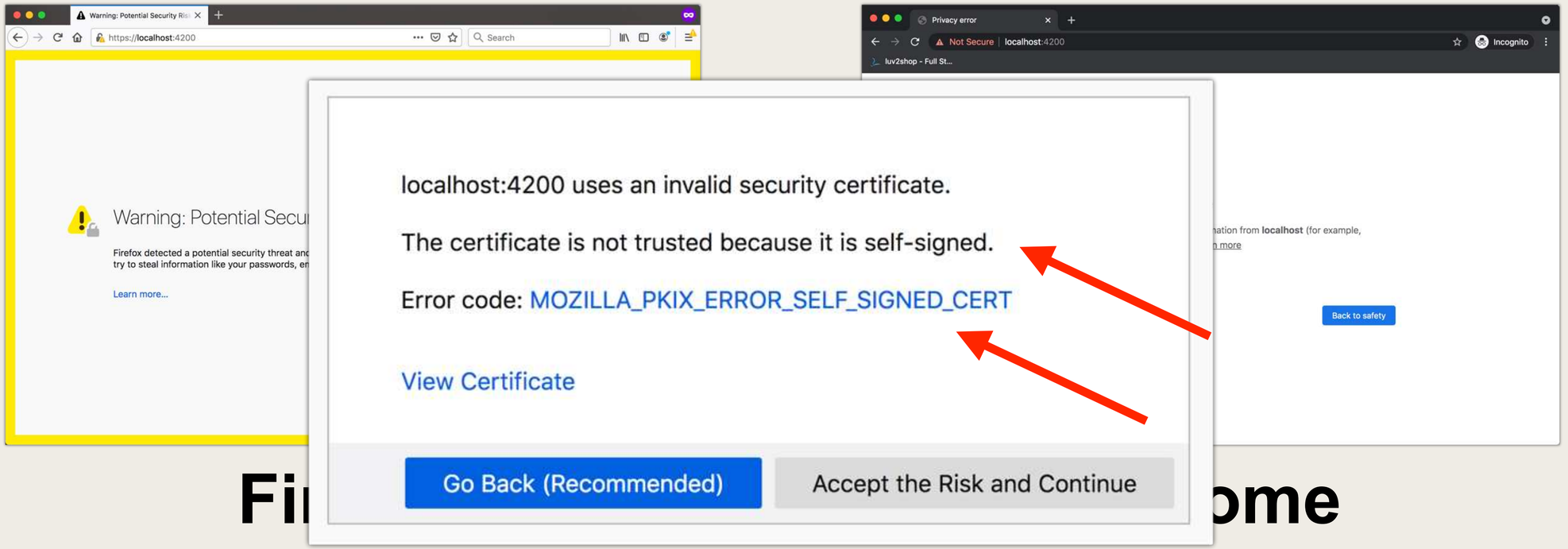
certificate

```
$ npm start
```

New command for starting our Angular app

Web Browser Warning

- Web browser will warn you when using self-signed certificates



Fi ome

Step 3: Update Spring Boot app with new URL

File: application.properties

```
...

allowed.origins=https://localhost:4200


...
```

Note the https

Since our Angular app will run using https

Additional Resources

- What is HTTPS, SSL, TLS?
- Keys and certificates

www.luv2code.com/https-additional-resources

Secure Communication
Okta Updates

Okta Updates

- Since Angular app is running on a different protocol
  - `https://localhost:4200`

- Need to update Okta configs for Redirect URIs

Development Process

1. Update Redirect URI in Angular App
2. Update Redirect URIs in Okta Dashboard
3. Update API Trusted Origins in Okta Dashboard

Step 1: Update Redirect URI in Angular App

File: my-app-config.ts

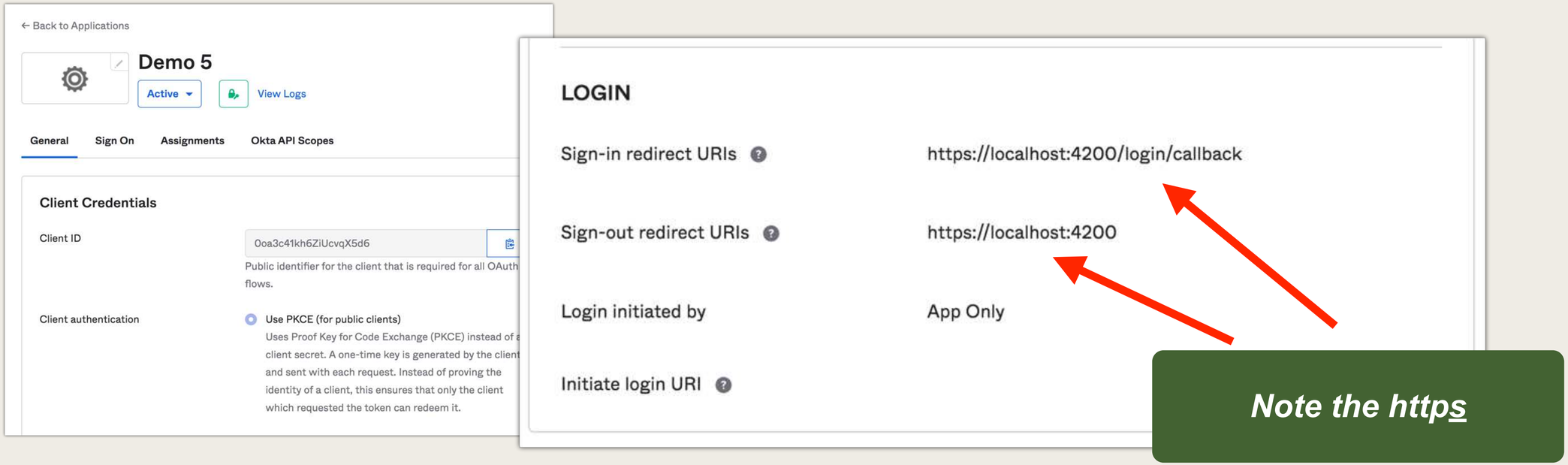
```
export default {

    oidc: {
        clientId: '<<your-client-id>>>',
        issuer: 'https://<<your-dev-domain>>/oauth2/default',
        redirectUri: 'https://localhost:4200/login/callback',
        scopes: ['openid', 'profile', 'email']
    }

}
```

Note the https

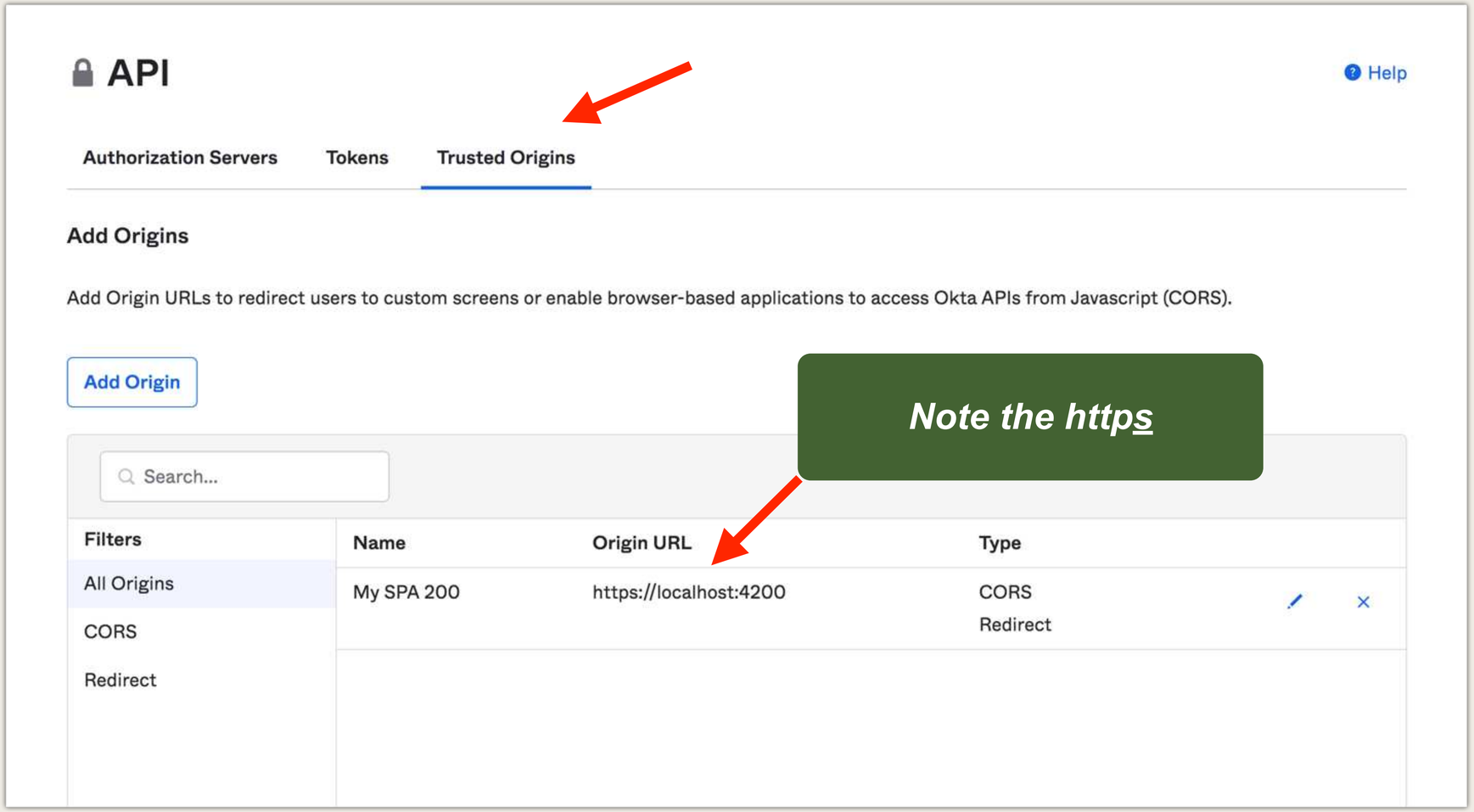
Since our Angular app will run using https

Step 2: Update Redirect URIs in Okta Dashboard

- Update redirect URIs



Step 3: Update API Trusted Origins in Okta Dashboard



Secure Communication with HTTPS

Secure Communication

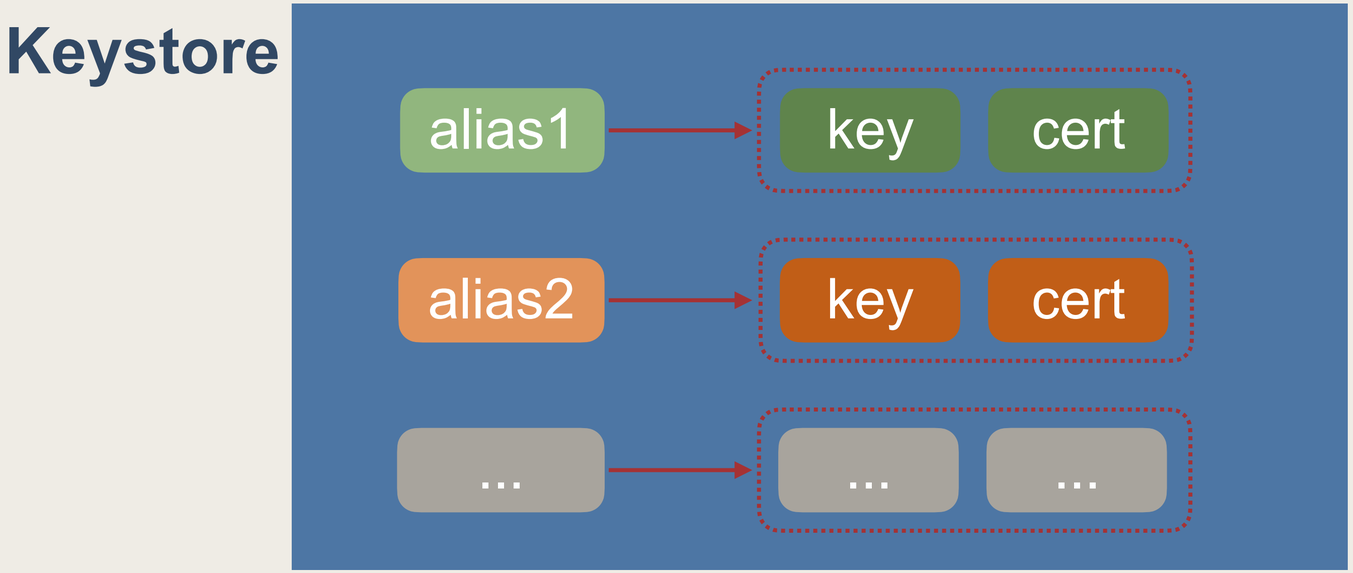
- We have secured Angular frontend
- Now let's secure the Spring Boot backend

Development Process

1. Generate key and self-signed certificate
2. Update application.properties with security configs

Java Keystore

- Java provides support for keys and certificates
- A keystore is a file that contains keys and certificates
- The entries are associated with an alias and password



Step 1: Generate key and self-signed certificate

- Use the JDK utility: `keytool`

```
$ keytool -genkeypair -noprompt \
  -alias luv2code \
  -keypass secret \
  -keystore src/main/resources/luv2code-keystore.p12 \
  -storeType PKCS12 -storepass secret \
  -keyalg RSA -keysize 2048 -validity 365 \
  -dname "C=US, ST=Pennsylvania, L=Philadelphia, O=luv2code, OU=Training Backend, CN=localhost" \
  -ext "SAN=dns:localhost"
```

Location of Java keystore file

Key info: alg, size, valid for 365 days

www.luv2code.com/java-keytool-docs

Step 2: Update application.properties with security configs

File: application.properties

```
# Server web port
server.port=8443

# Enable HTTPS support (only accept HTTPS requests)
server.ssl.enabled=true

# Alias that identifies the key in the key store
server.ssl.key-alias=luv2code

# Keystore location
server.ssl.key-store=classpath:luv2code-keystore.p12

# Keystore password
server.ssl.key-store-password=secret

# Keystore format
server.ssl.key-store-type=PKCS12
```

Based on keytool command in previous slide

```
$ keytool -genkeypair -noprompt \
  -alias luv2code \
  -keypass secret \
  -keystore src/main/resources/luv2code-keystore.p12 \
  -storeType PKCS12 -storepass secret \
  -keyalg RSA -keysize 2048 -validity 365 \
  -dname "C=US, ST=Pennsylvania, L=Philadelphia, O=luv2code, OU=Tra
  -ext "SAN=dns:localhost"
```

Angular Environments

We have a problem

- Our Angular app currently has Spring Boot URL hard coded

```
export class ProductService {

  private baseUrl = 'http://localhost:8080/api/products';

  private categoryUrl = 'http://localhost:8080/api/product-category';

  ...
```

- The Spring Boot backend is not even at the location anymore
  - New HTTPS location - **https://localhost:8443/api/products**

We have a problem

- Our Angular app currently has Spring Boot URL hard coded

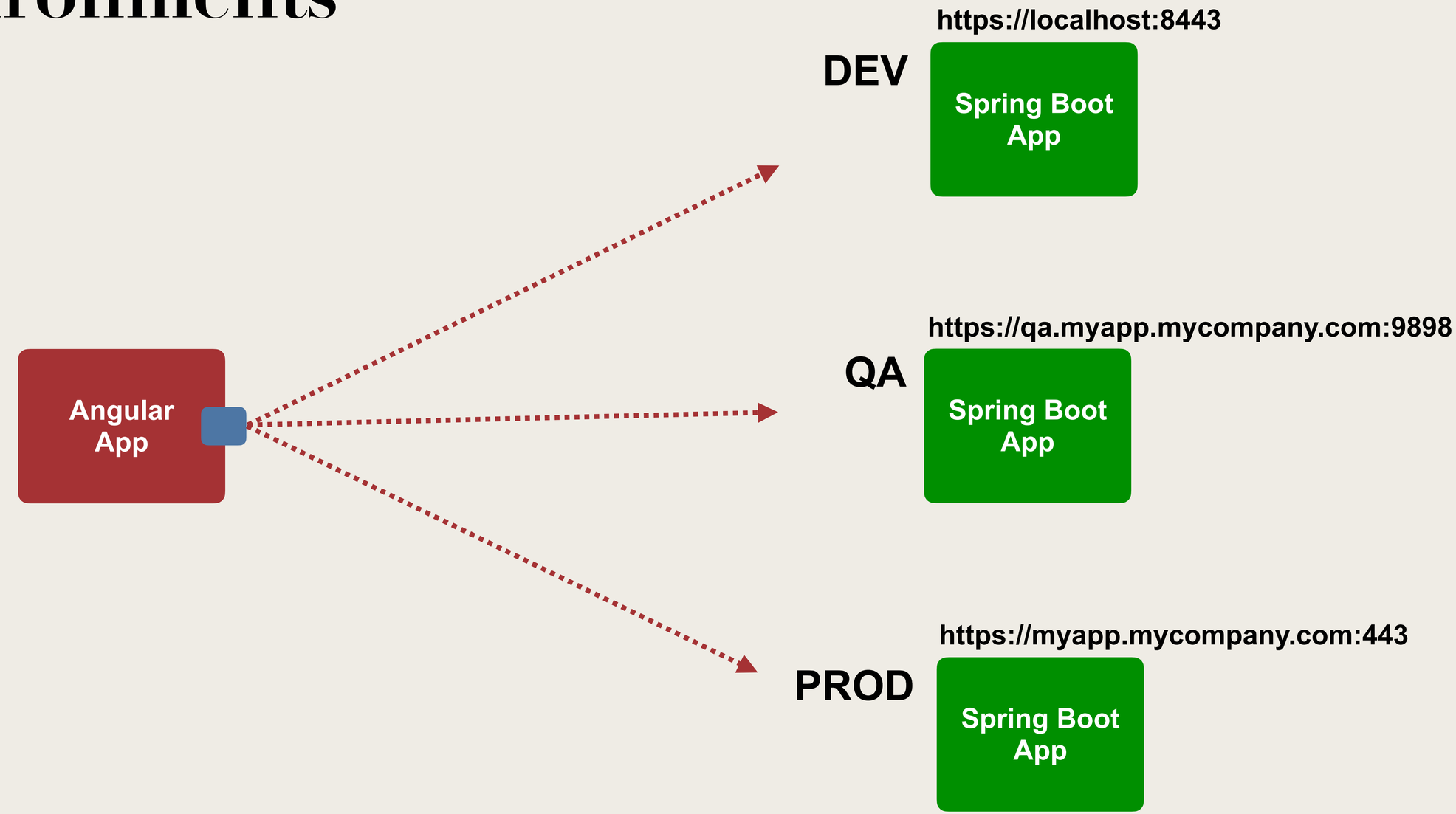
- The Spring Boot backend is not even at the location anymore
  - New HTTPS location - **https://localhost:8443/api/products**

Environments

- Instead of hard-coding URL in service class, place in a configuration
  - This will eliminate the need to change multiple references

- May need to use a different URL based on the environment
  - Allow app to easily run if deployed to different environment / server

Environments



Angular Environments

- Angular has support for **environments**
  - An environment is a named configuration for your app
  - You can add configurations for various environments

Angular Environments

- When you create a new Angular project usin
- By default, it creates the following files:
  - `src/environments`
    - `environment.ts`
    - `environment.prod.ts`

Default environment

Prod environment

Can add other environment files.
Can give any environment name.

```
environment.qa.ts
environment.betatest.ts
environment.westcoast.ts
...
```

environment.ts

```ts
export const environment = {
  production: false

  // add your custom environment configs
  // ...
};
```

environment.prod.ts

```ts
export const environment = {
  production: true

  // add your custom environment configs
  // ...
};
```

Development Process

1. Define configs in environment file
2. Use environment in your app
3. Run with environment configuration

Step 1: Define configs in environment file

environment.ts

```
export const environment = {
  production: false,

  // add your custom environment configs

  luv2shopApiUrl: "https://localhost:8443/api"

};
```

Define any name / value pairs

Step 2: Use environment in your app

Import the environment ... from Step 1

product.service.ts

```
import { environment } from 'src/environments/environment';


export class ProductService {

  private baseUrl = environment.luv2shopApiUrl + '/products';

  private categoryUrl = environment.luv2shopApiUrl + '/product-category';

  ...
}
```

Use the environment

environment.ts

```
export const environment = {
  production: false,

  // add your custom environment configs

  luv2shopApiUrl: "https://localhost:8443/api"

};
```

Step 3: Run with environment configuration

- By default, two environment configurations are defined: default and production

- Run with the default, just give command

```
$ npm start
```

Use environment.ts

- Run with production configuration

```
$ npm start -- --configuration=production
```

Use environment.prod.ts

Custom environments

- You can add custom environments and give any environment name

```
environment.qa.ts
environment.betatest.ts
environment.westcoast.ts
...
```

- Additional configuration steps are needed
- I'll cover this in upcoming videos :-)

Angular Environments

Add New Environment

Angular Environments

- In previous videos, we saw the use of environment files

- `src/environments`
  - `environment.ts`
  - `environment.prod.ts`

Default environment

environment.ts

```
export const environment = {
  production: false

  // add your custom environment configs
  // ...
};
```

Prod environment

environment.prod.ts

```
export const environment = {
  production: true

  // add your custom environment configs
  // ...
};
```

Run with environment configuration

- Run with the default, just give command

```
$ npm start
```

Use environment.ts

- Run with production configuration

```
$ npm start -- --configuration=production
```

Use environment.prod.ts

How does it work behind the scenes?

How are the filenames mapped to an environment?

How can I add a new environment?

Behind the scenes

- Environment configurations are defined in: `angular.json`
- Angular CLI tool added entries for "production" configuration in the file initially

angular.json

```
...
"configurations": {
  "production": {
    "fileReplacements": [
      {
        "replace": "src/environments/environment.ts",
        "with": "src/environments/environment.prod.ts"
      }
    ]
  }
}
...
```

Configuration name

During processing, for production configuration, will load the file: environment.prod.ts

Behind the scenes

- In the "serve" section of angular.json, defi

```
...
"configurations": {
  "production": {
    "fileReplacements": [
      {
        "replace": "src/environments/environment.ts",
        "with": "src/environments/environment.prod.ts"
      }
    ]
  }
}
...
```

angular.json

```
...
"serve": {
  "configurations": {
    "production": {
      "browserTarget": "angular-ecommerce:build:production"
    }
  }
},
...
```

Configuration name

Referenced from command-line when we start the app:

```
npm start -- --configuration=production
```

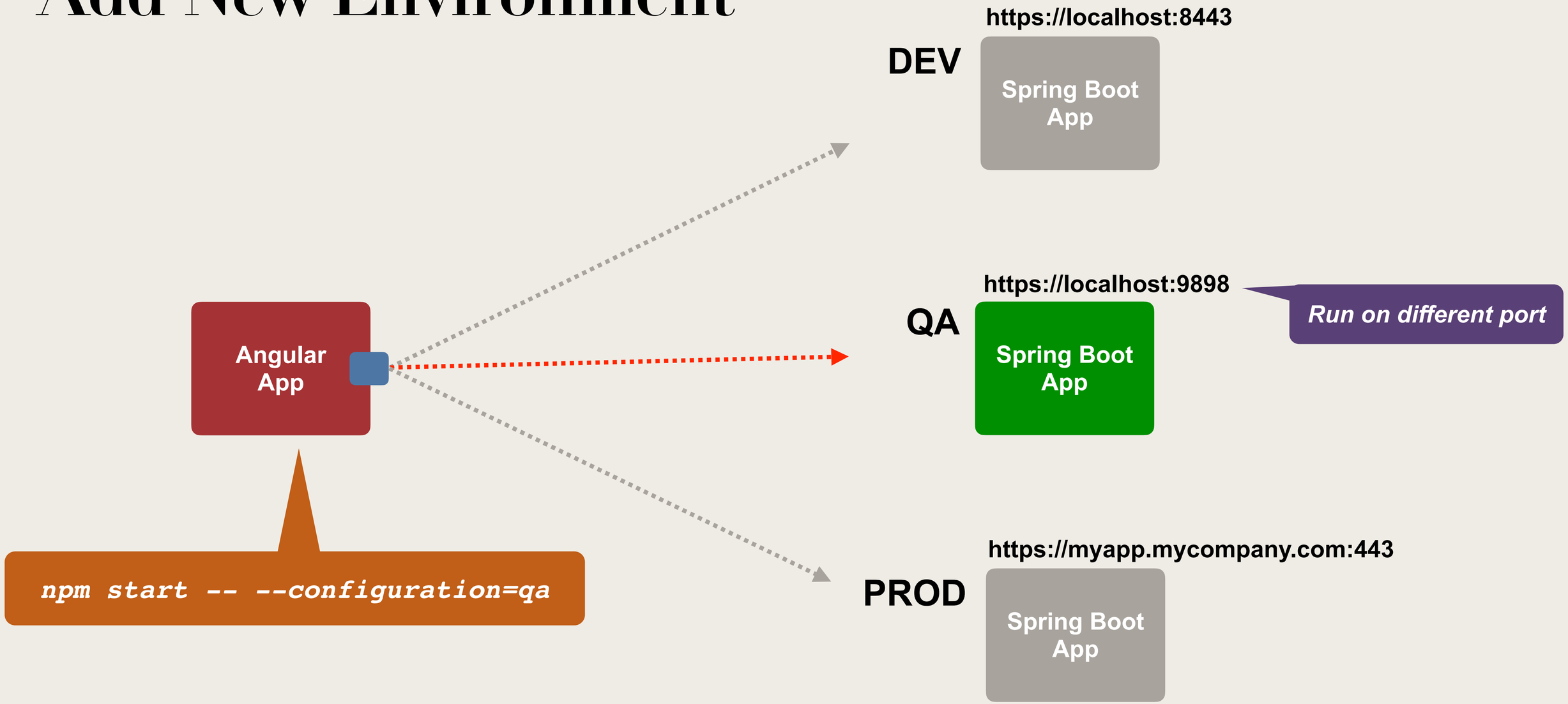
References the configuration defined on previous slide

~~How does it work behind the scenes?~~

~~How are the filenames mapped to an environment?~~

How can I add a new environment?

Add New Environment



Development Process

1. Create environment file: environment.qa.ts
2. Define configuration in angular.json for "build" section
3. Define configuration in angular.json for "serve" section
4. Run with environment configuration

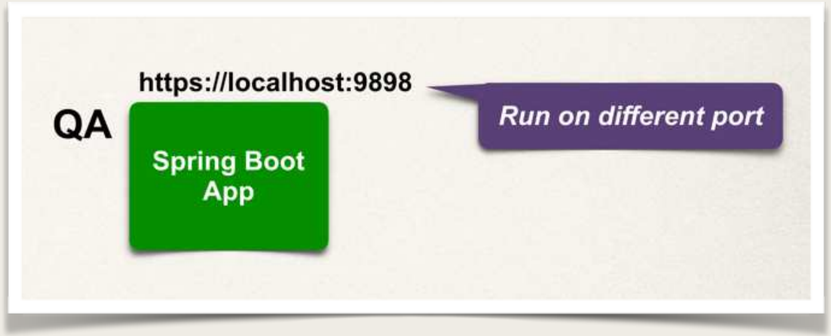
Step 1: Create environment file: environment.qa.ts

environment.qa.ts

```
export const environment = {
  production: false,

  // add your custom environment configs

  luv2shopApiUrl: "https://localhost:9898/api"

};
```

Same name

URL for Spring Boot app running in QA



Step 2: Define configuration in angular.json for "build" section

Path in angular.json file:
projects > angular-ecommerce > architect > build > configurations

angular.json

```
...
"configurations": {
  ...
  "qa": {
    "fileReplacements": [
      {
        "replace": "src/environments/environment.ts",
        "with": "src/environments/environment.qa.ts"
      }
    ]
  }
}
```

Configuration name

Can use any file name

Best practice:
Stay consistent with configuration name

Makes it easy for other developers to understand your project

During processing, for qa configuration, will load the file: environment.qa.ts

Step 3: Define configuration in angular.json for "serve" section

Path in angular.json file:
projects > angular-ecommerce > architect > serve > configurations

angular.json

```
...
"serve": {
  "configurations": {
    "qa": {
      "browserTarget": "angular-ecommerce:build:qa"
    }
  }
},
...
```

Configuration name

```
...
"configurations": {
  "qa": {
    "fileReplacements": [
      {
        "replace": "src/environments/environment.ts",
        "with": "src/environments/environment.qa.ts"
      }
    ]
  }
}
...
```

Referenced from command-line when we start the app:

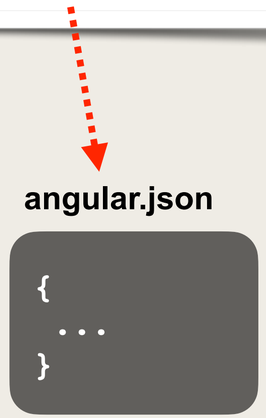
```
npm start -- --configuration=qa
```

References the configuration defined on previous slide

Step 4: Run with environment configuration

- Run with qa configuration

```
$ npm start -- --configuration=qa
```



environment.qa.ts

```
export const environment = {
  production: false,

  // add your custom environment configs

  luv2shopApiUrl: "https://localhost:9898/api"

};
```

Why production: false, production: true???

environment.ts

```
export const environment = {
  production: false,
  ...
};
```

environment.prod.ts

```
export const environment = {
  production: true,
  ...
};
```

main.ts

if true

```
...
if (environment.production) {
  enableProdMode();
}
...
```

Prod Mode

- Reduce internal logging messages
- Disable file checks for auto-reloading
- Minify scripts and styles
- Eliminate dead-code
- Inline critical CSS and fonts
- ....

Additional Resources

www.luv2code.com/angular-optimization-configuration