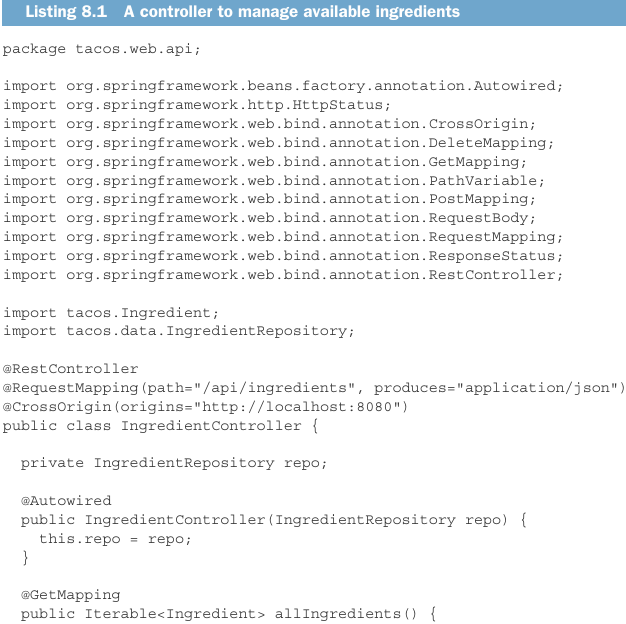
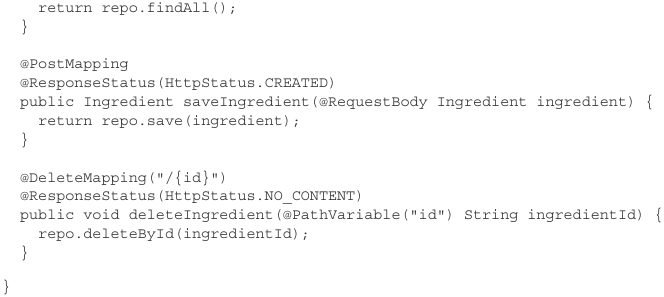
8. Securing REST

-Securing a REST API is different from securing a browser-based web application.

# 8.1 Introducing OAuth 2

-Suppose we want to create a new back-office application to manage Taco Cloud app. Before that, we need to add new endpoints to Taco Cloud API to support ingredient management.



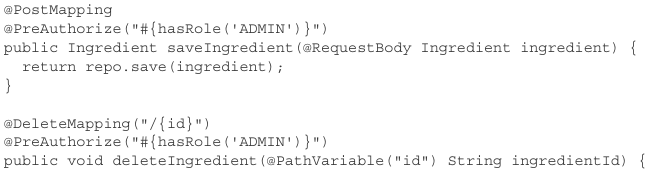




+But there’s no security around that API. Anyone can use

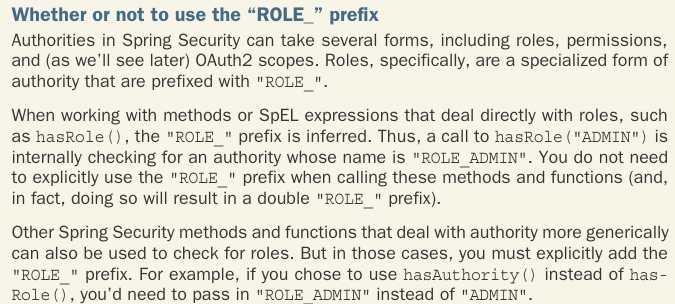
-API is part of main app and available to the world. GET endpoint is used by user interface in home.html. We need to secure at least POST and DELETE endpoints

+We can use **HTTP Basic authentication** by add **@PreAuthorize**

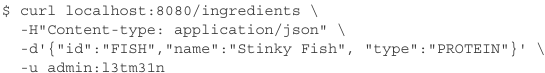


+Or in **security configuration**:



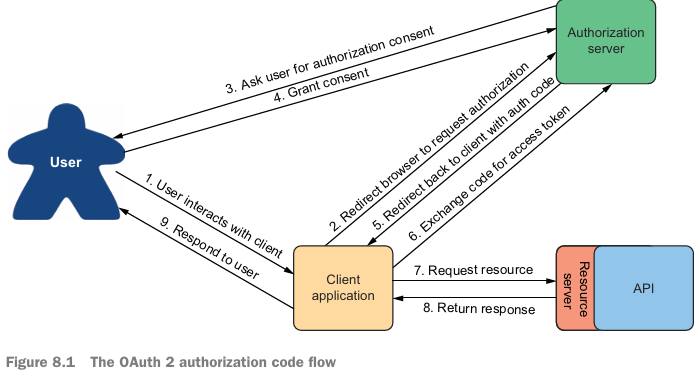


-Either way, the ability so submit these requests will require that the submitter provide credentials that have “ROLE\_ADMIN” authority: use -u

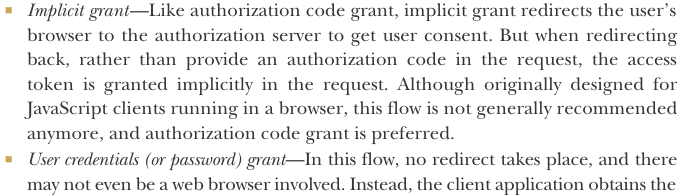


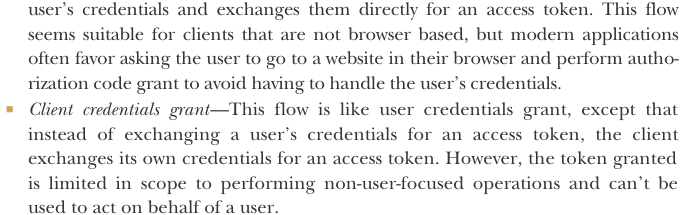
-Instead of requiring the admin user identify on every request, API just asks for some **token** that proves they are authorized to access the resource. It’s like a ticket to a sporting event.

->That’s how OAuth 2 authorization works: Clients request an access token (like a valet key) from an authorization server with the express permission of a user. That token allows them to interact with API on behalf of the user who authorized the client. At any point, the token could expire or be revoked (without requiring user’s password change). In such cases, the client needs to request a new access token:

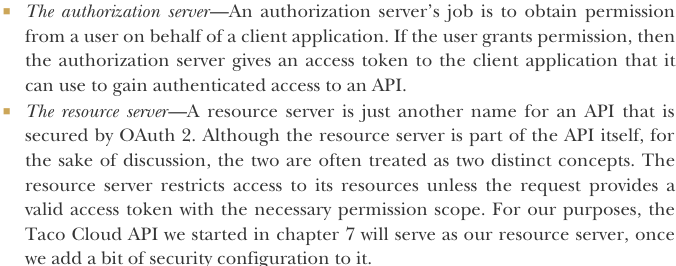


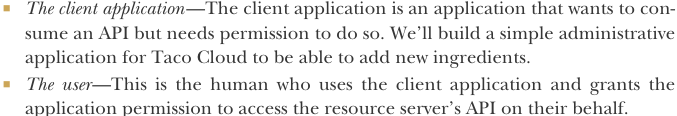
-**OAuth 2** is very rich security specification that offers lots of ways to use it. The flow above is **authorization code grant.** Other flow supported by OAuth 2:





-For our purpose, we focus on **authorization code grant flow** to obtain **JSON Web Token (JWT)** access token. This involves creating a handful of applications that work together:





-You can **dig deeper** into the subject by reading **OAuth 2** **specification** (<https://oauth.net/2/>) or these books:

+**OAuth 2 in Action**: https://www.manning.com/books/oauth-2-in-action +**Microservices Security in Action**: https://www.manning.com/books/microservices-security-in-action

+**API Security in Action**: <https://www.manning.com/books/api-security-in-action>

+liveProject “Protecting User Data with Spring Security” and OAuth2 (<http://mng.bz/4KdD>)

-Developers are encouraged to use authorization severs from vendors: Okta, Google… Spring Authorization Server project is a great way to get started with OAuth 2.

-We going to see how to use **OAuth 2** using **Spring Security**. We’ll create **2 new project**: an **authorization server project** + a **client project**, modify **existing Taco Cloud project** such that its API acts as a **resource server**.

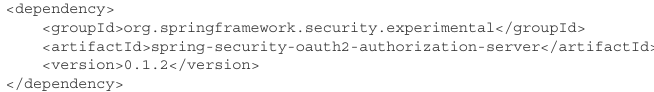
# 8.2 Creating an authorization server

-Authorization server’s job it to issue an access token on behalf of a user.

-Spring Authorization Server is experimental and doesn’t implement all OAuth 2 grant types but it does implement authorization code grand + client credentials grant.

-Authorization server is a distinct app from any app that provides API and from the client. -> Create a new SB project: web + security + jpa+lombok. [Spring Initializr](https://start.spring.io/)

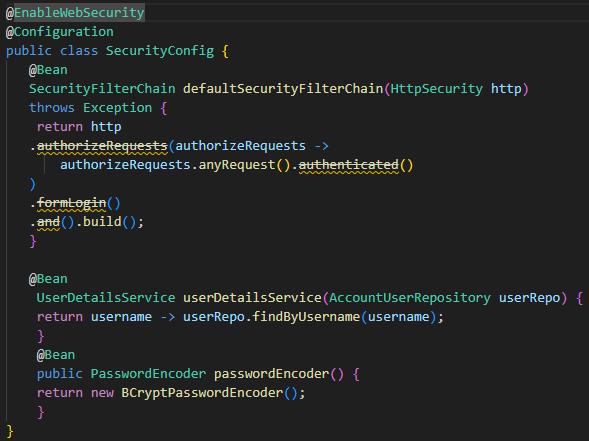
-Manually add SAS dependency:



-Make sure that there’s not a port conflict

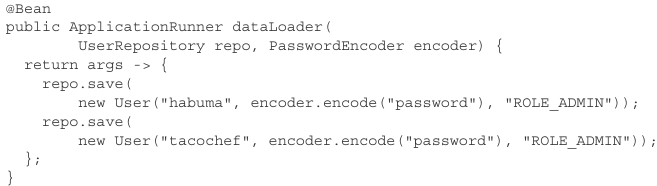


-Simple SS configuration class that enables form-based login and requires that all requests be authenticated



+Notice: UserDetailsService bean works with TacoUserRepository, you need to specifics User + UserRepository like we’ve created in Chap 3.

-You can use CommandLineRunner bean to prepopulate database with some test users:

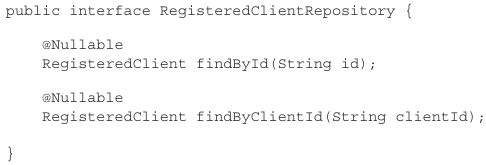


-Apply **configuration** to **enable authorization server**.

+**authorizationServerSecurityFilterChain()** bean: set up default behavior for OAuth 2 authorization server + default form login page.

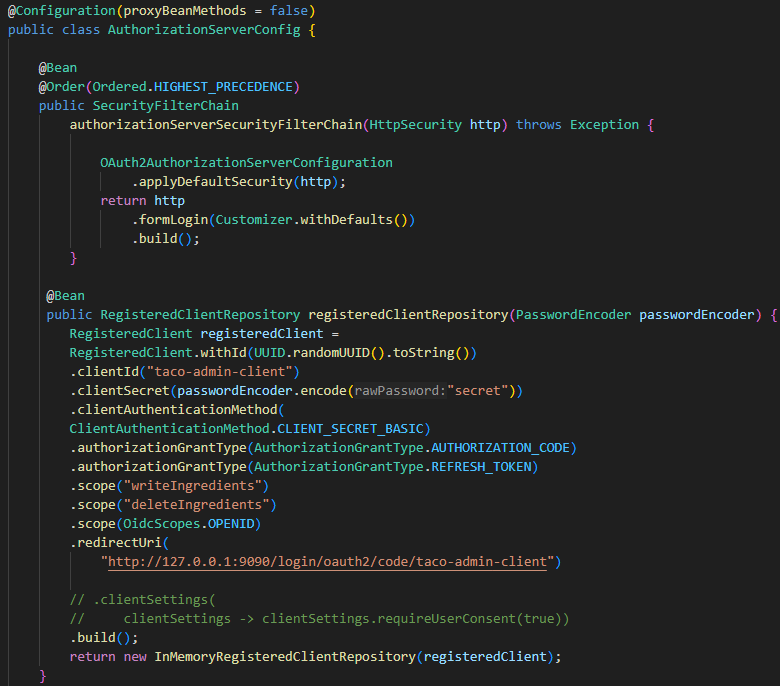
**+@Order** with **Ordered.HIGHEST\_PRECEDENCE**: ensure that if for some reason there are other beans of this type declared, this one takes precedence over the others.

-**Client repository**: maintains details about clients that might be asking for authorization on behalf of users. It’s defined by **RegisteredClientRepository** interface:

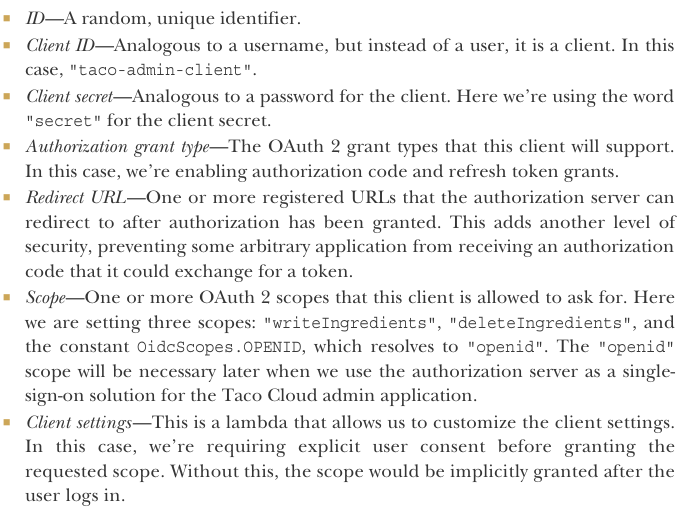


+In production setting, you can write a custom implementation of RegisteredClientRepository to retrieve client details from database or from some other source.

-SAS offers in-memory implementation for demonstration and testing purposes in authorization configuration:



-Here’s how our client is defined:



-Finally, our authorization server will be producing JWT tokens, the tokens need to include a signature creatd using JSON Web Key (JWK) <https://datatracker.ietf.org/doc/html/rfc7517> as the signing key.



+To summarize: JWKSource creates RSA 2048-bit key pairs that will be used to sign the token. The token will be signed using the private key. The resource server can then verify that token received in a request is valid by obtaining the public key form authorization server.

-Build and run the app, you should have an authorization server listening on port 9000.

+Test in web browser: localhost:9000/oauth2/authorize?response\_type=code&client\_id=tacoadmin-client& redirect\_uri=http://127.0.0.1:9090/login/oauth2/code/taco-admin-client&-scope=write Ingredients+deleteIngredients

After that, the browser will be directed to client URI

+

# 8.3 Securing an API with a resource server

# 8.4 Developing the client

# -Summary