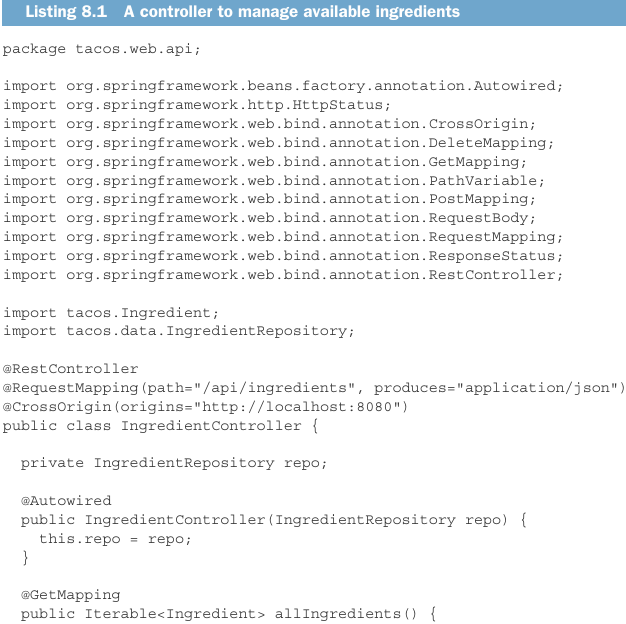
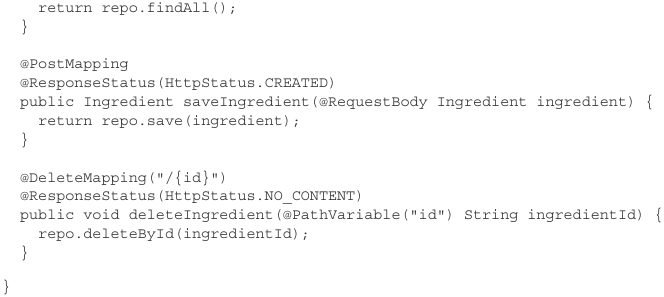
8. Securing REST

-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.



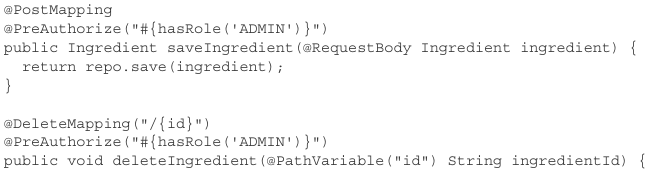




+But there’s no security around that API. Anyone can use

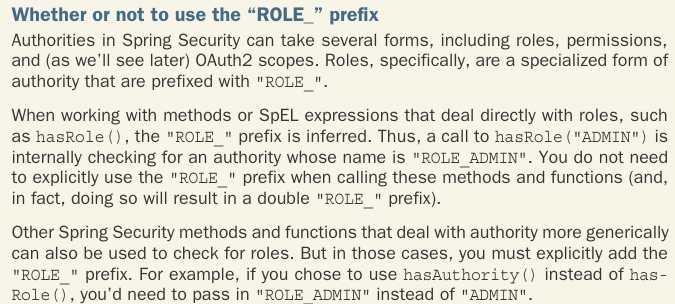
-API is part of main app and available to the world. GET endpoint is used by user interface in home.html. We need to secure at least POST and DELETE endpoints

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

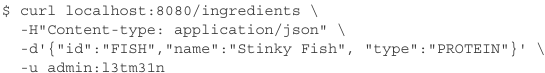


+Or in **security configuration**:



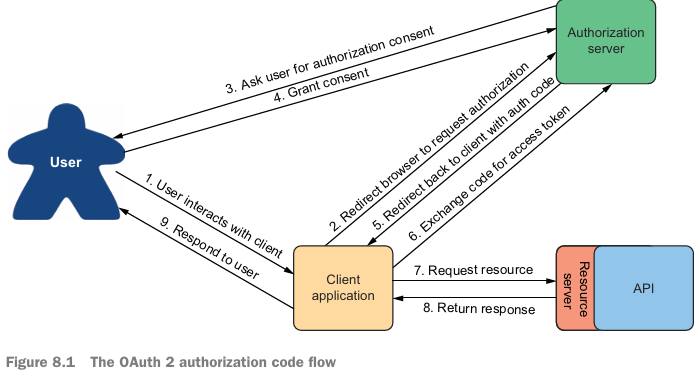


-Either way, the ability so submit these requests will require that the submitter provide credentials that have “ROLE\_ADMIN” authority: use -u

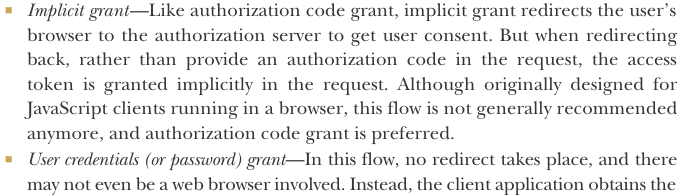


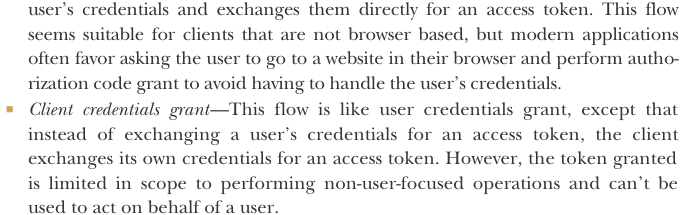
-Instead of requiring the admin user identify on every request, API just asks for some **token** that proves they are authorized to access the resource. It’s like a ticket to a sporting event.

->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:

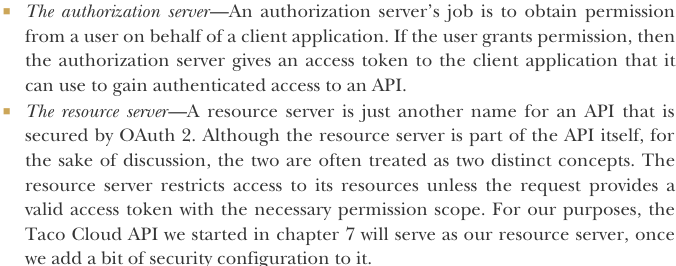


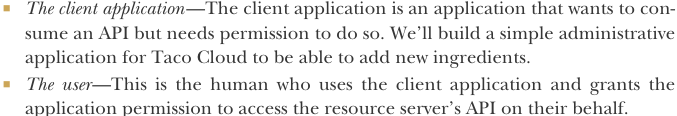
-**OAuth 2** is very rich security specification that offers lots of ways to use it. The flow above is **authorization code grant.** Other flow supported by OAuth 2:





-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**.

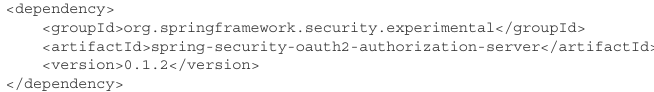
# 8.2 Creating an authorization server

-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/)

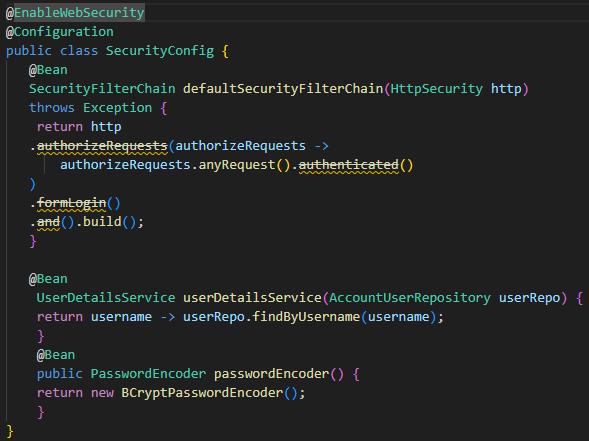
-Manually add **SAS dependency**:



-Make sure that there’s not a port conflict

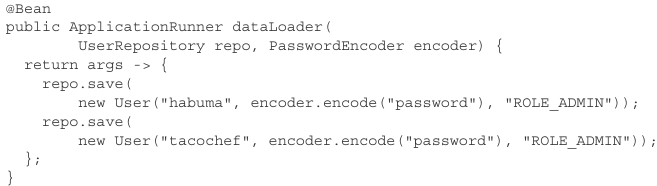


-Simple SS configuration class that enables form-based login and requires that all requests be authenticated



+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:

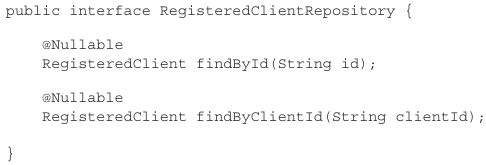


-Apply **configuration** to **enable authorization server**.

+**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.

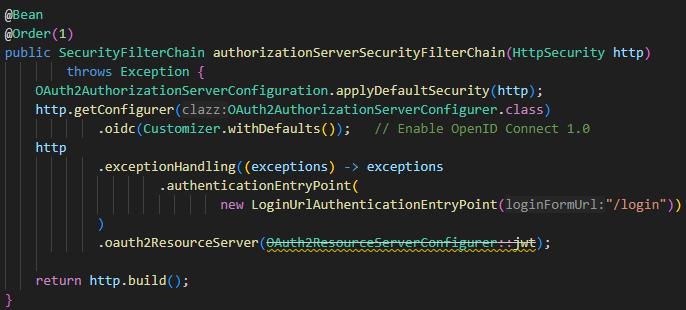
-**Client repository**: maintains details about clients that might be asking for authorization on behalf of users. It’s defined by **RegisteredClientRepository** interface:

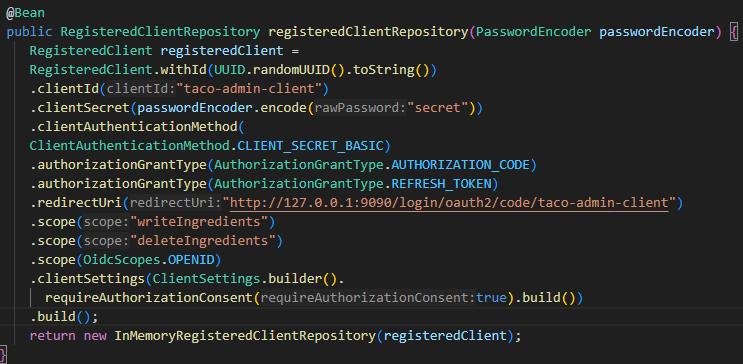


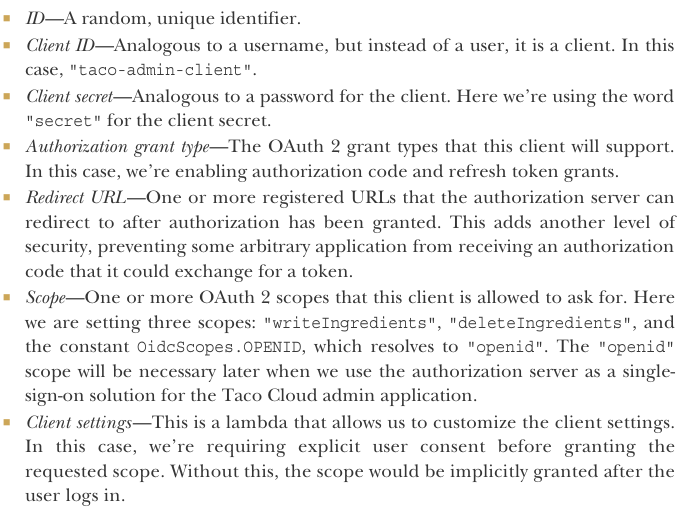
+In production setting, you can write a custom implementation of RegisteredClientRepository to retrieve client details from database or from some other source.

-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.



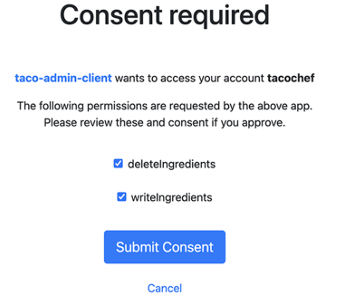
+To summarize: JWKSource creates RSA 2048-bit key pairs that will be used to sign the token. The token will be signed using the private key. The resource server can then verify that token received in a request is valid by obtaining the public key form authorization server.

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

+Test in web browser:

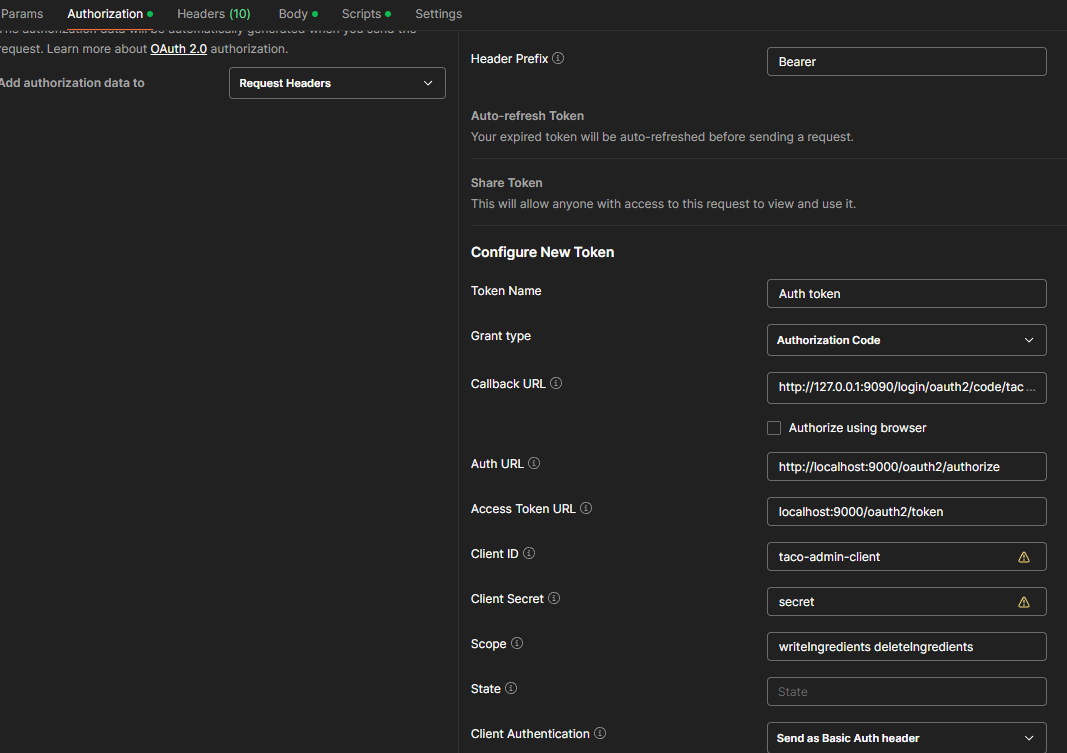
<http://localhost:9000/oauth2/authorize?response_type=code&client_id=taco-admin-client&redirect_uri=http://127.0.0.1:9090/login/oauth2/code/taco-admin-client&scope=writeIngredients+deleteIngredients>

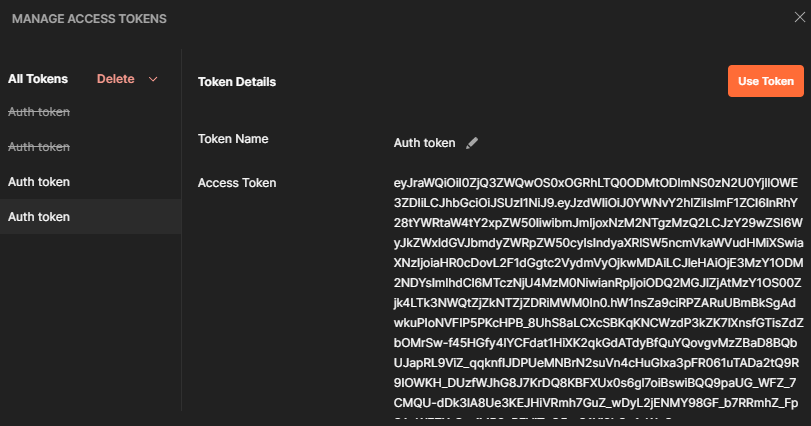
-After loggin in (us username-password combination in the database under TacoUserRepository), you’ll be asked to consent to requested scopes:



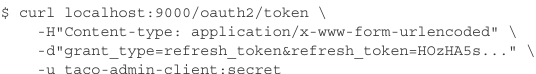
After that, the browser will be directed to client URI

-Using Postman:



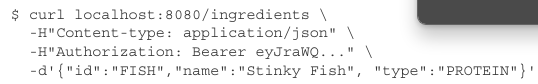


-The access token can be sent on requests the resource server to write or delete Ingredients. This token will expire in 299 seconds. If it expires, we can use refresh token to obtain new access token:

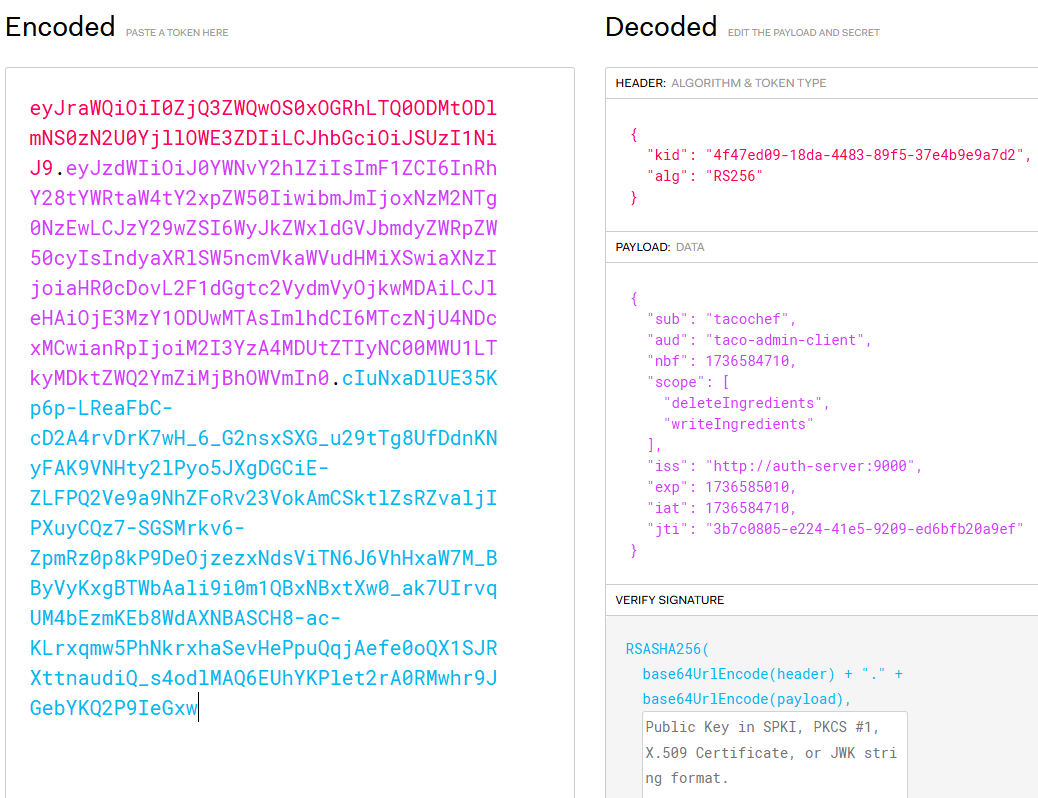


Postman: 

-To use access token, send it in a request to Taco Cloud API as part of “Authorization” header:



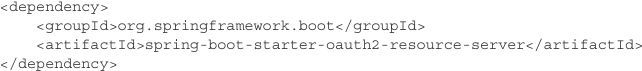
-We can inspect the access token by copy + past into the form of <https://jwt.io/>



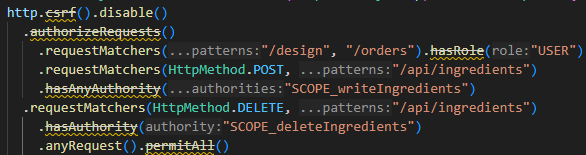
+payload: show that this token was issued on behalf of the user and scopes.

# 8.3 Securing an API with a resource server

-SS provides an OAuth2 resource server implementation that add to an existing API:

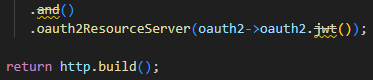


-Declare POST requests /ingredients require “writeIngredients” scope + DELETE requests /ingredietns require “deleteIngredients” scope:



+hasAuthority(): specify the required scope. Scopes are prefixed with “SCOPE\_” to indicate that the should be matched against OAuth 2 scopes in access token.

-In that same configuration class, Also enable resource server:

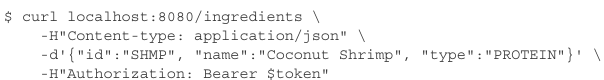


-Configure the resource server to know where to obtain the public key:

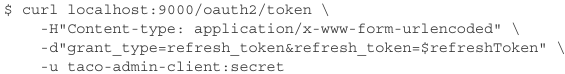


-Build Taco Cloud app:

+Provide access token in Authorization header: substituting $token for actual access token



+Get new access token when the token expire:



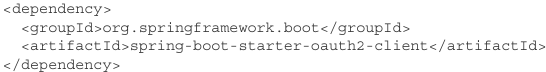
-Now we can apply the same techniques to secure other sensitive endpoints in API: /orders

# 8.4 Developing the client

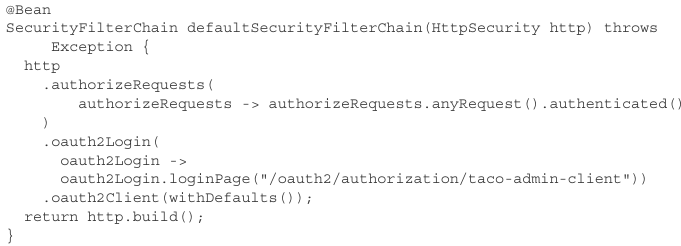
-The client’s role: obtain an access token + make request to resource server on behalf of the user.

-Using OAuth2’s authorization code flow, when the client app determines that the user has not yet been authenticated, it should redirect the user’s browser to authorization server to get consent from user. When authorization server redirects control back to client, the client must exchange the authorization code it receives for access token.

-Add Spring Security’s OAuth 2 client dependency:

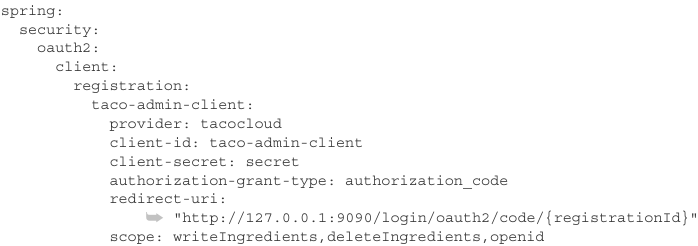


-Require authentication:



+Enable the client-side bits of OAuth2: Set up a login page at path /oauth2/authorization/taco-admin-client. It accepts an authorization code, exchange it for an access token and uses it to determine the identity of user. This is the path that the authorization server will redirect to after the user has granted permission.

-Configure about the authorization server and applicatoin’s OAuth 2 client:



+This registers a client with Spring Security OAuth2 taco-admin-client. The registration details: client’s credentials, grant type, scope, redirect URI. The value in redirect uri has a placeholder that references client’s registration ID

-provider property: set to tacocloud, which is a reference to a separate set of configuration that describes the tacocloud provider’s authorization server.



+issuer-uri: the base URI for authorization server. On most Unix-based OS, this can be added in your/etc/hosts file: 

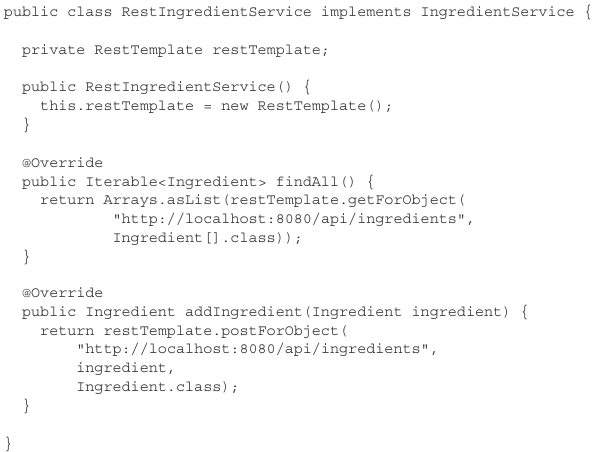
-If the authorization server differs from default values, explicitly configure authorization details:



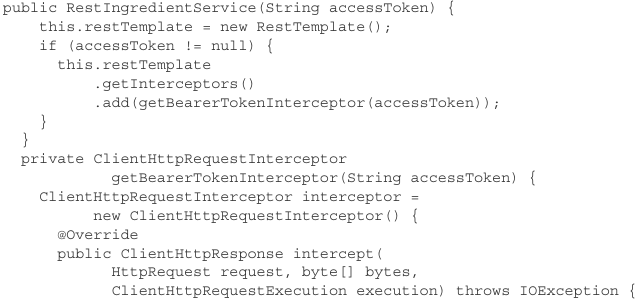
+user-info-uri: used by client to obtain essential user information.

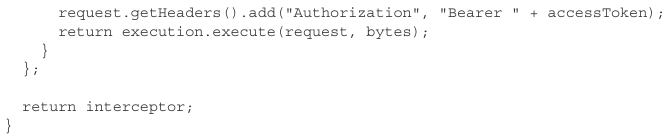
+user-name-attribute: property A request to user-info-uri should return JSON response

-Suppose we have a service bean that interact with Taco Cloud API using RestTemplate:



-Spring Security’s OAuth2 client should have access token after completing authorization code flow. The access token ends up in the request: change constructor to attach a request interceptor to RestTemplate





+The constructor take a String that is the access token. It attaches a client request interceptor that adds Authorization header to every request made by RestTemplate such that header’s value is “Bearer” followed by token value.

-Where does the access token come from. In config:



+@RequestScope: a new instance of bean will be created on every request. The been needs to pull the authentication from SecurityContext, which is populated on every request by one of SS’s filters.

+Before return RestIngredientService instance, the bean method checks the authentication is implemented as OAuth2AuthenticationToken. Then verifies the authentication token is for client taco-admin-client. Then it extracts the token from authorized client and passes it through constructor of RestIngredientService ->make requests to Taco Cloud API’s endpoints on behalf of user

# -Summary

