Lab: Simple Spring Security

Important!: You can download initial project for this lab from GitHub: `https://github.com/fenago/spring-boot-java/tree/main/labs/lab_7`

1: Understand our Security Requirements

Who should be allowed to manage any given Cash Card?

In our simple domain, let's state that the user who created the Cash Card "owns" the Cash Card. Thus, they are the "card owner". Only the card owner can view or update a Cash Card.

The logic will be something like this:

IF the user is authenticated

... AND they are authorized as a "card owner"

... ... AND they own the requested Cash Card

THEN complete the users's request

BUT do not allow users to access Cash Cards they do not own.

2: Review update from Previous Lab

In this lab we'll secure our Family Cash Card API and restrict access to any given Cash Card to the card's "owner".

To prepare for this, we introduced the concept of an owner in the application.

The owner is the unique identity of the person who created and can manage a given Cash Card.

Let's review the following changes we made on your behalf:

owner added as a field to the CashCard Java record. owner added to all .sql files in src/test/resources/ owner added to all .json files in src/test/resources/example/cashcard All application code and tests are updated to support the new owner field. No functionality has changed as a result of these updates.

Let's take some time now to familiarize yourself with these updates.

3: Add the Spring Security Dependency

We can add support for Spring Security by adding the appropriate dependency.

Add the dependency.

Add the following to the build.gradle file in the dependencies {} section:

dependencies { implementation 'org.springframework.boot:spring-boot-starter-web'

```
// Add the following dependency
implementation 'org.springframework.boot:spring-boot-starter-security'
...
```

Run the tests.

We've added Spring Security capabilities to our application, but changed no code.

So what do we expect to happen when we run the tests?

Note that we will always run ./gradlew test to run the tests.

[~/exercises] $ ./gradlew test ... CashCardApplicationTests > shouldReturnASortedPageOfCashCards() FAILED ... CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() FAILED ... CashCardApplicationTests > shouldCreateANewCashCard() FAILED ... CashCardApplicationTests > shouldReturnAPageOfCashCards() FAILED ... CashCardApplicationTests > shouldReturnAllCashCardsWhenListIsRequested() FAILED ... CashCardApplicationTests > shouldReturnASortedPageOfCashCardsWithNoParametersAndUseDefaultValues() FAILED ... CashCardApplicationTests > shouldNotReturnACashCardWithAnUnknownId() FAILED 11 tests completed, 7 failed

> *Task :test FAILED Things are really broken!*

Every test method within CashCardApplicationTests failed.

Many failures are similar the the one below:

```
expected: <SOME NUMBER>
 but was: 0
```

In most cases, our tests expect CashCard data to be returned from our API, but nothing was returned.

Why do you think all tests of our Cash Card API are failing after adding the Spring Security dependency?

Understand why everything is broken.

So what happened?

When we added the Spring Security dependency to our application, security was enabled by default.

Since we have not specified how authentication and authorization are performed within our Cash Card API, Spring Security has completely locked down our API.

Better safe than sorry, right?

Next, let's configure Spring Security for our application.

4: Satisfy Spring Security's Dependencies

Next, we'll focus on getting our tests passing again by providing the minimum configuration needed by Spring Security.

We've provided another file on our behalf: `example/cashcard/SecurityConfig.java`. This will be the Java Bean where we'll configure Spring Security for our application.

Uncomment `SecurityConfig.java` and review.

Open SecurityConfig.

Notice that most of the file is commented.

Uncomment all commented lines within SecurityConfig.

```
package example.cashcard;
...

public class SecurityConfig {

    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http.build();
...
```

filterChain returns http.build(), which is the minimum needed for now.

Note: Please ignore the method passwordEncoder() for now.

Enable Spring Security.

At the moment SecurityConfig is just an un-referenced Java class as nothing is using it.

Let's turn SecurityConfig into our configuration Bean for Spring Security.

// Add this Annotation @Configuration public class SecurityConfig {

```
// Add this Annotation
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http.build();
}
```

... Understand the Annotations.

@Configuration public class SecurityConfig {...} The @Configuration annotation tells Spring to use this class to configure Spring and Spring Boot itself. Any Beans specified in this class will now be available to Spring's Auto Configuration engine.

@Bean public SecurityFilterChain filterChain Spring Security expects a Bean to configure its Filter Chain, which you learned about in the Simple Spring Security lesson. Annotating a method returning a SecurityFilterChain with the @Bean satisfies this expectation.

Run the tests.

When you run the tests you'll see that once again all tests pass except for the test for creating a new CashCard via a POST.

CashCardApplicationTests > shouldCreateANewCashCard() FAILED org.opentest4j.AssertionFailedError: expected: 201 CREATED but was: 403 FORBIDDEN ... 11 tests completed, 1 failed This is expected. We'll cover this in depth a bit later on.

5: Configure Basic Authentication

Thus far we have bootstrapped Spring Security, but not actually secured our application.

Now let's secure our application by configuring basic authentication.

Configure basic authentication.

Update SecurityConfig.filterChain with the following to enable basic authentication:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  http.authorizeHttpRequests()
    .requestMatchers("/cashcards/**")
    .authenticated()
    .and()
    .csrf().disable()
    .httpBasic();
  return http.build();
}
```

Understand the Spring Security configuration.

That's a lot of method calls!

Here if we explain Spring Security's builder pattern in more understandable language, we see:

All HTTP requests to cashcards/ endpoints are required to be authenticated using HTTP Basic Authentication security (username and password).

Also, do not require CSRF security.

Note: We'll talk about CSRF security later in this lab.

Run the tests.

What will happen when we run our tests?

When you run the tests, you'll notice that most tests fail with a 401 UNAUTHORIZED HTTP status code, such as the following:

expected: 200 OK but was: 401 UNAUTHORIZED Though it might not look like it, this is progress!

We've enabled basic authentication requiring that requests must supply a username and password.

Our tests do not provide a username and password with our HTTP requests. So let's do that next.

6: Testing Basic Authentication

As we learned in the accompanying lesson, there are many ways of providing user authentication and authorization information for a Spring Boot application using Spring Security.

For our tests, we'll configure a test-only service that Spring Security will use for these this purpose: an InMemoryUserDetailsManager.

Similar to how we configured an in-memory database using H2 for testing Spring Data, we'll configure an in-memory service with test users to test Spring Security.

Configure a test-only UserDetailsService.

Which username and password should we submit in our test HTTP requests?

When you reviewed changes to src/test/resources/data.sql you should've seen that we set an OWNER value for each CashCard in the database to the username sarah1. For example:

INSERT INTO CASH_CARD(ID, AMOUNT, OWNER) VALUES (100, 1.00, 'sarah1'); Let's provide a test-only UserDetailsService with the user sarah1.

Add the following Bean to SecurityConfig.

```
@Bean
public UserDetailsService testOnlyUsers(PasswordEncoder passwordEncoder) {
 User.UserBuilder users = User.builder();
 UserDetails sarah = users
   .username("sarah1")
   .password(passwordEncoder.encode("abc123"))
   .roles() // No roles for now
   .build();
 return new InMemoryUserDetailsManager(sarah);
}
```

This UserDetailsService configuration should be understandable: configure a user named sarah1 with password abc123.

Spring's IoC container will find the UserDetailsService Bean and Spring Data will use it when needed.

Configure Basic Auth in HTTP tests.

Select one test method that uses restTemplate.getForEntity and update it with basic authentication for sarah1.

```
void shouldReturnACashCardWhenDataIsSaved() {
    ResponseEntity<String> response = restTemplate
            .withBasicAuth("sarah1", "abc123") // Add this
            .getForEntity("/cashcards/99", String.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    ...
```

Run the tests.

The updated test that provides the basics should now pass!

... CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() PASSED ... Update all remaining CashCardApplicationTests tests and rerun tests.

Now for some tedium: update all remaining restTemplate-based tests to supply .withBasicAuth("sarah1", "abc123") with every HTTP request.

When finished, rerun the test.

```
BUILD SUCCESSFUL in 9s
```

Everything passes!

Congratulations, you've implemented and tested Basic Auth!

Verify Basic Auth with additional tests.

Now let's add tests that expect a 401 UNAUTHORIZED response when incorrect credentials are submitted using basic authentication.

```
@Test
void shouldNotReturnACashCardWhenUsingBadCredentials() {
    ResponseEntity<String> response = restTemplate
      .withBasicAuth("BAD-USER", "abc123")
      .getForEntity("/cashcards/99", String.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.UNAUTHORIZED);

    response = restTemplate
      .withBasicAuth("sarah1", "BAD-PASSWORD")
      .getForEntity("/cashcards/99", String.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.UNAUTHORIZED);
}
```

This test should pass...

```
...
CashCardApplicationTests > shouldNotReturnACashCardWhenUsingBadCredentials() PASSED
```

Success! Now that we've implemented authentication, let's move on to implement authorization next.

7: Support Authorization

As we learned in the accompanying lesson, Spring Security supports many forms of authorization.

Here we'll implement Role-Based Access Control (RBAC).

It is likely that a user service will provide access to many authenticated users, but only "card owners" should be allowed to access Family Cash Cards managed by our application. Let's make those updates now.

Add a users and roles to the UserDetailsService Bean.

To test authorization, we need multiple test users with a variety of roles.

Update SecurityConfig.testOnlyUsers and add the CARD-OWNER role to sarah1.

Also, let's add a new user named "hank-owns-no-cards" with a role of NON-OWNER.

```
...
@Bean
public UserDetailsService testOnlyUsers(PasswordEncoder passwordEncoder) {
  User.UserBuilder users = User.builder();
  UserDetails sarah = users
    .username("sarah1")
    .password(passwordEncoder.encode("abc123"))
    .roles("CARD-OWNER") // new role
    .build();
  UserDetails hankOwnsNoCards = users
    .username("hank-owns-no-cards")
    .password(passwordEncoder.encode("qrs456"))
    .roles("NON-OWNER") // new role
    .build();
  return new InMemoryUserDetailsManager(sarah, hankOwnsNoCards);
}
```

Test for Role verification.

Let's add a test that will fail at first, but will pass when we fully implement authorization.

Here we'll assert that user "hank-owns-no-cards" should not have access to a CashCard since that user is not a CARD-OWNER.

@Test void shouldRejectUsersWhoAreNotCardOwners() { ResponseEntity response = restTemplate .withBasicAuth("hank-owns-no-cards", "qrs456") .getForEntity("/cashcards/99", String.class); assertThat(response.getStatusCode()).isEqualTo(HttpStatus.FORBIDDEN); } But wait! CashCard with ID 99 belongs to sarah1, right? Shouldn't only sarah1 have access to that data regardless of role?

You're right! Keep that in mind for later in this lab.

Run the tests.

We see that our new test fails when we run it.

```
CashCardApplicationTests > shouldRejectUsersWhoAreNotCardOwners() FAILED
 org.opentest4j.AssertionFailedError:
```

```
 expected: 403 FORBIDDEN
  but was: 200 OK
```

Why was hank-owns-no-cards able to access a CashCard as indicated by the 200 OK response?

Although we have given the test users roles, we are not enforcing role-based security.

Enable role-based security.

Edit SecurityConfig.filterChain to restrict access to only users with the CARD-OWNER role.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  http.authorizeHttpRequests()
    .requestMatchers("/cashcards/**")
    .hasRole("CARD-OWNER") // enable RBAC: Replace the .authenticated() code with this
line.
    .and()
    .csrf().disable()
    .httpBasic();
  return http.build();
}
```

Run the tests.

We see that our tests pass!

CashCardApplicationTests > shouldRejectUsersWhoAreNotCardOwners() PASSED We've now successfully enabled RBAC-based authorization!

8: Cash Card ownership: Repository Updates

As mentioned in the previous exercise, we have a glaring security hole in our application.

Any authenticated user with role CARD-OWNER can view anyone else's Family Cash Cards!

To fix this, we will update our tests, CashCardRepository, and CashCardController:

We'll add functionality to CashCardRepository to restrict or "scope" queries to the correct OWNER. Next, we'll update our CashCardController to guarantee that only the correct OWNER is used. Learning Moment: Best Practices Wait! Isn't this lab all about the Spring Security project and its amazing technology? Can't Spring Security do all of this ownership validation so we don't have to modify our Repositories and Controllers?

Answer: This lab is about securing an HTTP API. Yes, security technologies such as Spring Security are amazing. Yes, there are features such as Spring Security Method Security that might help in this situation, but it is still your responsibility as a developer to write secure code and follow best security practices. For example, don't write code that allows users to access other users' data!

Now, let's update our tests and CashCardRepository.

Add a new CashCard for a user named kumar2.

Update src/test/resources/data.sql with a CashCard record owned by a different user:

```
...
INSERT INTO CASH_CARD(ID, AMOUNT, OWNER) VALUES (102, 200.00, 'kumar2');
```

Test that users cannot access each other's data.

Let's add a test that explicitly asserts that our API returns a 404 NOT FOUND when a user attempts to access a Cash Card they do not own.

Note: You might wonder why we want to return a 404 NOT FOUND response instead of something else, like 401 UNAUTHORIZED. One argument in favor of choosing to return NOT FOUND is that it's the same response that we'd return if the requested Cash Card doesn't exist. It's safer to err on the side of not revealing any information about data which is not authorized for the user.

Now we'll have sarah1 attempt to access kumar2's data.

```
@Test
void shouldNotAllowAccessToCashCardsTheyDoNotOwn() {
    ResponseEntity<String> response = restTemplate
      .withBasicAuth("sarah1", "abc123")
      .getForEntity("/cashcards/102", String.class); // kumar2's data
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
}
```

Run the tests.

What do you think will happen when we run the tests? Let's find out.

As expected, our new test fails, along with many others.

```
CashCardApplicationTests > shouldNotAllowAccessToCashCardsTheyDoNotOwn() FAILED
 org.opentest4j.AssertionFailedError:
 expected: 404 NOT_FOUND
  but was: 200 OK
```

What's going on here?

Answer: Currently, user sarah1 is able to view kumar2's data because:

sarah1 is authenticated. sarah1 is an authorized CARD-OWNER.

In addition, our test for fetching a list of CashCards is also failing:

```
CashCardApplicationTests > shouldReturnAllCashCardsWhenListIsRequested() FAILED
 org.opentest4j.AssertionFailedError:
 expected: 3
  but was: 4
```

Why are we returning too many Cash Cards? For the same reason as above: sarah1 has access to kumar2's data. kumar2's Cash Card is being returned to sarah1's list of Cash Cards.

Let's prevent users from accessing each other's data.

Update the CashCardRepository with a new findById methods.

The simplest thing we can do is to always filter our data access by CashCard owner.

Let's do that in our Repository.

We will need to filter by owner when finding both a single CashCard or a list of CashCards.

Edit CashCardRepository to add a new finder methods.

Be sure to include the new imports for Page and PageRequest.

```
...
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
...

public interface CashCardRepository extends CrudRepository<CashCard, Long>,
PagingAndSortingRepository<CashCard, Long> {
   CashCard findByIdAndOwner(Long id, String owner);
   Page<CashCard> findByOwner(String owner, PageRequest amount);
}
```

As mentioned in the Spring Data labs and lessons, Spring Data will take care of the actual implementations (writing the SQL queries) for us.

Note: You might wonder whether Spring Data allows you to write your own SQL. After all, Spring Data can't anticipate every need, right? The answer is Yes! It's easy for you to write your own SQL code. The Spring Data Query Methods documentation describes how to do so by using the @Query annotation.

Spring Data is perfectly capable of generating the SQL for the queries we need, though. Thanks, Spring Data!

Next, let's update the Controller.

9: Cash Card ownership: Controller Updates

The CashCardRepository now supports filtering CashCard data by owner.

But we're not using this new functionality. Let's make those updates to the CashCardController now by introducing a concept we explained in the written lesson: the Principal.

As with other helpful objects, the Principal is available for us to use in our Controller. The Principal holds our user's authenticated, authorized information.

Update the Controller's GET by ID endpoint.

Update the CashCardController to pass the Principal's information to our Repository's new findByIdAndOwner method.

Be sure to add the new import statement.

```
import java.security.Principal;
...

@GetMapping("/{requestedId}")
public ResponseEntity<CashCard> findById(@PathVariable Long requestedId, Principal
principal) {
    Optional<CashCard> cashCardOptional =
Optional.ofNullable(cashCardRepository.findByIdAndOwner(requestedId,
principal.getName()));
    if (cashCardOptional.isPresent()) {
     ...
```

Note that Principal.name() will return the username provided from Basic Auth.

Run the tests.

The GET is passing, but our tests for Cash Card lists are failing.

CashCardApplicationTests > shouldReturnASortedPageOfCashCards() FAILED ... CashCardApplicationTests > shouldReturnACashCardWhenDataIsSaved() PASSED ... CashCardApplicationTests > shouldReturnAllCashCardsWhenListIsRequested() FAILED ... CashCardApplicationTests > shouldReturnASortedPageOfCashCardsWithNoParametersAndUseDefaultValues() FAILED ... Update the Controller's GET for lists endpoint.

Edit `CashCardController` to filter lists by owner.

```
@GetMapping
public ResponseEntity<List<CashCard>> findAll(Pageable pageable, Principal principal)
{
    Page<CashCard> page = cashCardRepository.findByOwner(principal.getName(),
            PageRequest.of(
                pageable.getPageNumber(),
                ...
```

Once again we get the authenticated username from the principal.getName() method.

Run the tests.

They all pass!

```
BUILD SUCCESSFUL in 8s
```

10: Cash Card ownership: Creation Updates

We have one more remaining security hole: creating CashCards.

The authenticated, authorized Principal should be used as the owner when creating a new CashCard.

Question: What would happen if we automatically used the submitted owner value?

Answer: We risk allowing users to create CashCards for someone else!

Let's ensure that only the authenticated, authorized Principal owns the CashCards they are creating.

Update the POST test.

To prove that we do not need to submit an owner, let's use null as the owner for the CashCard.

```
void shouldCreateANewCashCard() {
  CashCard newCashCard = new CashCard(null, 250.00, null);
  ...
```

Run the tests.

What do you think will happen when we run the tests? They will likely fail, but can you guess why?

```
CashCardApplicationTests > shouldCreateANewCashCard() FAILED
  org.opentest4j.AssertionFailedError:
  expected: 201 CREATED
  but was: 500 INTERNAL_SERVER_ERROR
```

Our application is crashing due to the missing owner which is required by the database.

Review `test/resources/schema.sql` to see more.

```
CREATE TABLE cash_card
(
  ...
  OWNER    VARCHAR(256) NOT NULL
);
```

Update the POST endpoint in the Controller.

Once again we will use the provided Principal to ensure that the correct owner is saved with the new CashCard.

```
@PostMapping
private ResponseEntity<Void> createCashCard(@RequestBody CashCard newCashCardRequest,
UriComponentsBuilder ucb, Principal principal) {
    CashCard cashCardWithOwner = new CashCard(null, newCashCardRequest.amount(),
principal.getName());
    CashCard savedCashCard = cashCardRepository.save(cashCardWithOwner);
    ...
```

Run the tests.

Once again everything passes!

```
CashCardApplicationTests > shouldCreateANewCashCard() PASSED
...
BUILD SUCCESSFUL in 7s
```

Now only the authenticated, authorized Principal is used to create a CashCard.

Security for the win!

11: About CSRF

As we learned in the accompanying lesson, protection against Cross-Site Request Forgery (CSRF, or "sea-surf") is an important aspect of HTTP-based APIs used by web-based applications.

Yet, we've disabled CSRF via the .csrf().disable() line in SecurityConfig.filterChain.

Why have we disabled CSRF?

For the purposes of our Family Cash Card API, we're going to follow the guidance from the Spring Security team regarding non-browser clients:

When should you use CSRF protection? Our recommendation is to use CSRF protection for any request that could be processed by a browser by normal users. If you are only creating a service that is used by non-browser clients, you will likely want to disable CSRF protection.

If you would like to add CSRF security to our application, please review the testing support options below.

MockMVC CSRF testing examples WebTestClient CSRF testing examples. A description of the Double-Submit Cookie Pattern. The Cash Card codebase, with CSRF protection enabled and implementing tests using the Double-Submit Cookie pattern: TestRestReplate CSRF testing examples.

12: Summary

In this lab you learned how to use Spring Security to ensure that only authenticated, authorized users have access to the Family Cash Card API. In addition, you followed best practices in the Controller and Repository layers of our application to ensure that only the correct users have access to their (and only their) Cash Card data.