# Lab testen Spring boot met JUnit en Mockito

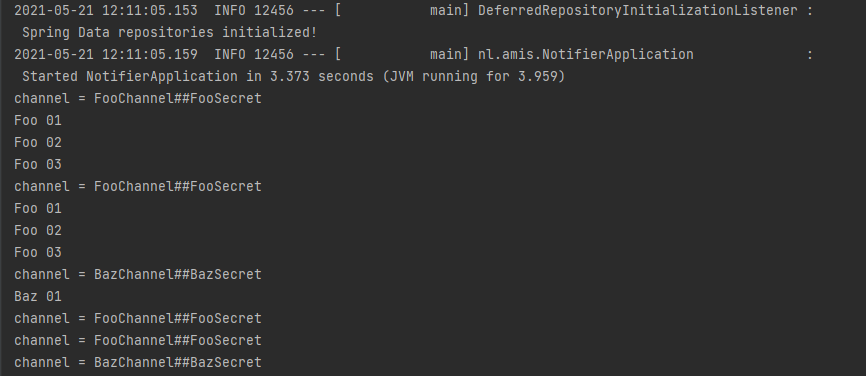
A good reference on writing different kinds of tests for Springboot applications see:<https://howtodoinjava.com/spring-boot2/testing/>

# Lab1: NotifierApplication project

At <https://github.com/AMIS-Services/sig-java-unittesting> you can download a working Java 11 + Spring boot application called NotifierApplication. The program contains logic for subscribing to notification channels and publishing notification messages to the channels that have been subscribed upon.

At startup an in memory database is provisioned with some subscriptions on message channels. Also some notification messages are written to database.

As can be seen in the output that is written/logged to the console the notifications are published per subscriber.



## In memory database

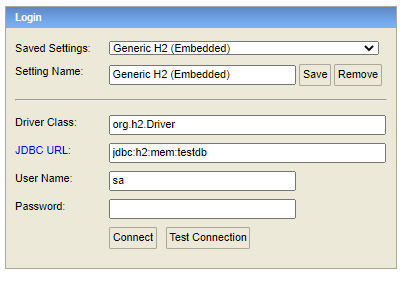
After the application has started running you can open a console for viewing and manipulating the contents in the in memory database.

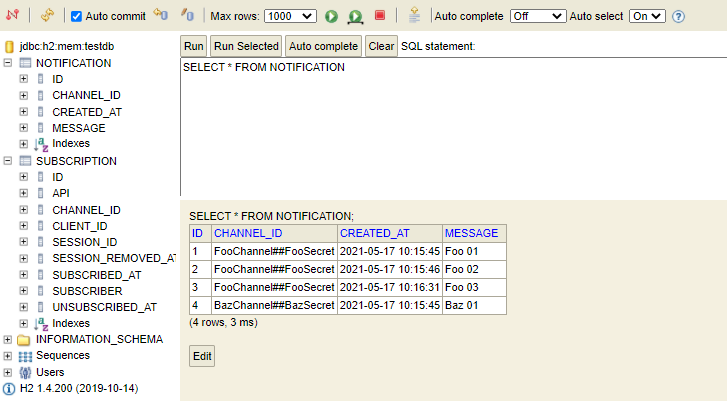
Navigate to the following url:

**http://localhost:8080/ h2-console**

Username: sa

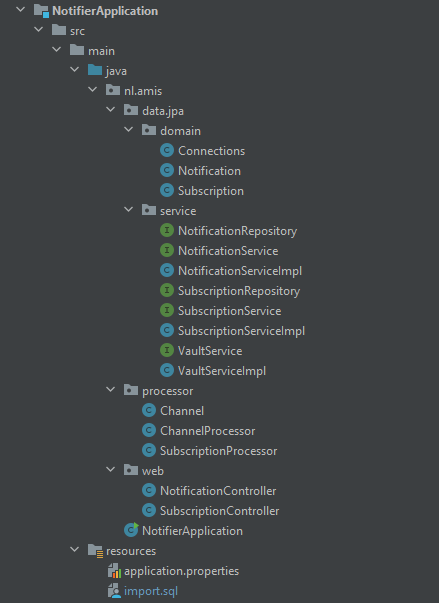
Password: password





## Project structure

The structure of the NotifierApplication is as follows:



The folder nl.amis.data.jpa contains the classes used for getting data in and out of the database.

Of main interest are the classes in the folder nl.amis.processor.

SubscriptionProcessor fetches active subscriptions from the database. For these subscriptions it starts a sendNotifications process that will be invoked every 15 seconds.

The sendNotification process does three things:

* It finds newly added subscriptions and binds a new processor (a ChannelProcessor that handles the notifications for the subscription) for each
* It finds the ended/removed subscriptions and removes the processor bound to it
* It invokes the processors bound to the subscriptions

ChannelProcessor fetches the notifications for a specific channel from database and publishes these to the console. As mentioned a channelprocessor is instantiated and bound to each active subscription.

# Pom.xml

## JaCoCo plugin

**Code coverage** is [a software metric](https://www.baeldung.com/cs/code-coverage) used to measure how many lines of our code are executed during automated tests. For measuring and reporting on the code coverage of the tests JaCoCo will be used.

* As an initial setup in the project, add the jacoco plugin to the pom.xml file in the build section (as below).

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 <plugin>  
 <groupId>org.jacoco</groupId>  
 <artifactId>jacoco-maven-plugin</artifactId>  
 <version>0.8.4</version>  
 <executions>  
 <execution>  
 <goals>  
 <goal>prepare-agent</goal>  
 </goals>  
 </execution>  
 <execution>  
 <id>report</id>  
 <phase>prepare-package</phase>  
 <goals>  
 <goal>report</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

## Setup Junit 5 dependencies

* First exclude Junit 4 from the spring-boot starter-test dependency in the pom.xml as follows:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 <exclusions>  
 <exclusion>  
 <groupId>org.junit.vintage</groupId>  
 <artifactId>junit-vintage-engine</artifactId>  
 </exclusion>  
 </exclusions>  
</dependency>

* Then add the dependency for Junit 5

<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter</artifactId>  
 <version>5.7.2</version>  
 <scope>test</scope>  
</dependency>

# Writing Junit tests

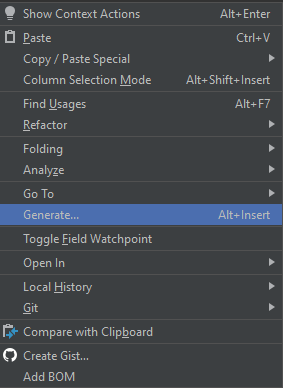
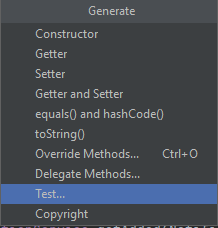
Now you are ready to write some unit tests, for which you will be using the Junit 5 framework.

* Navigate to the SubscriptionProcessor.java file and open it.

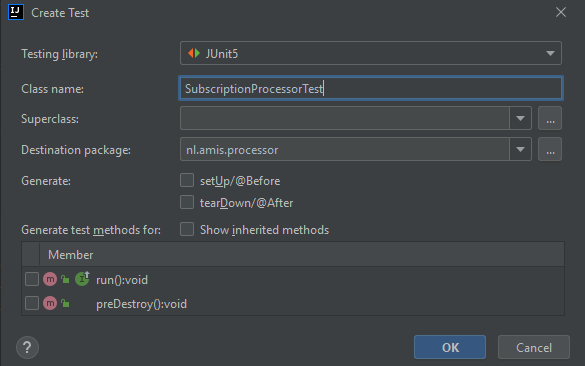
IntelliJ: Somewhere in the opened file right click and select Generate (or use the shortcut Alt+Insert)

**In IntelliJ IDE:**

* Select the method and right click on it. From the popup menu select Generate option and then Test...

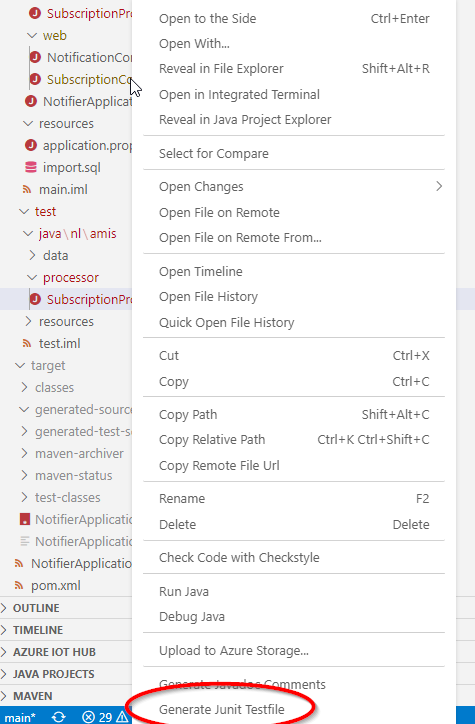
 

Now a new window is opened for creating the tests



**In Visual Studio Code:**

To auto-generate Junit files in Visual Studio Code, you can use the plugin: Junit Testfile Generator. Right-click on the file and select: Generate Junit Testfile. The options here are more limited than in IntelliJ.



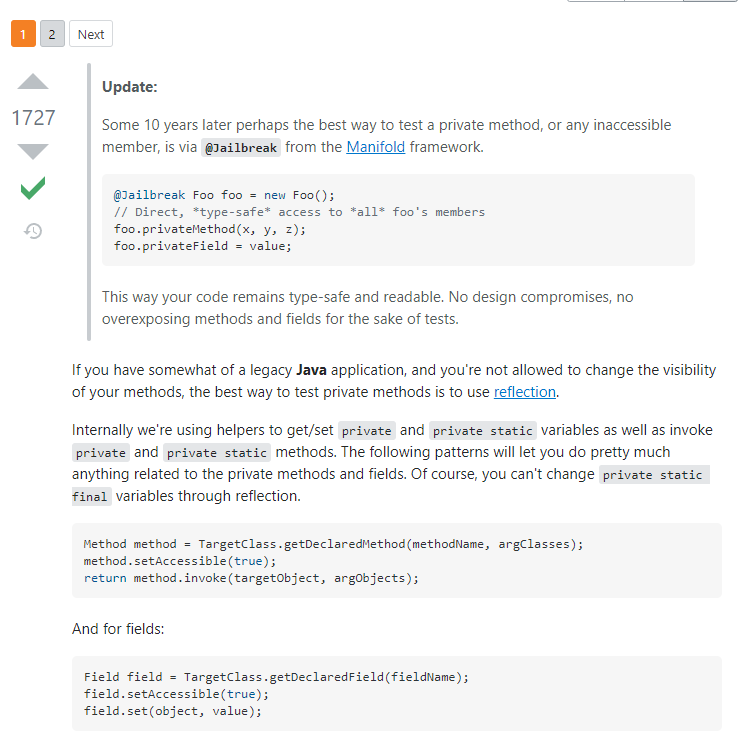
By default (as you can see below the Members), only the public methods are available for creating tests. This is an annoyance which can be dealt with in a few different ways:

* Don't test private methods.
* **Give the methods package access.**
* Use a nested test class.
* Use reflection.

For details see: [artima - Testing Private Methods with JUnit and SuiteRunner](https://www.artima.com/articles/testing-private-methods-with-junit-and-suiterunner)

More recent possibilities:

* Use @JailBreak from the Manifold framework
* Use reflection



* Make private methods package visible in the class SubscriptionProcessor.java

To be able to test the methods from another test class the private visibility has to be adjusted to package private level. For clearance though add some comment indicating the reason for this alteration, e.g.:

//private visibility changed to package visibility for testing purposes.

~~private~~ getProcessorsForActiveSubscriptions()

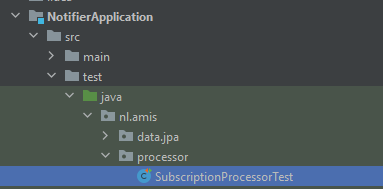
//private visibility changed to package visibility for testing purposes.

~~private~~ Map<String, Channel> determineChannelMap(final Iterable<Subscription> subscriptions) {

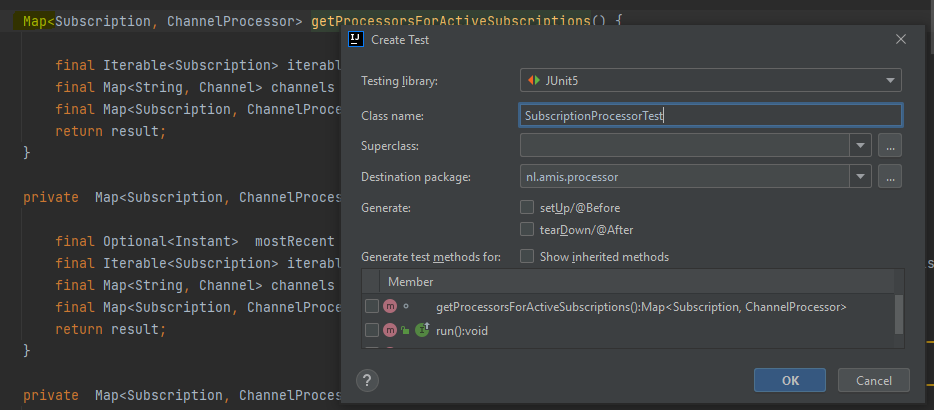
//private visibility changed to package visibility for testing purposes.

~~private~~ Map<Subscription, ChannelProcessor> createChannelProcessors(final Map<String, Channel> channels, final Iterable<Subscription> iterable) {

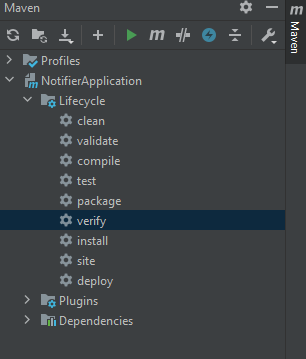
* Generate a test for this method (ALT-INSERT -> Generate -> Test… -> select the option getProcessorsForActiveSubscriptions(..).
* Because this is the first test also select the setUp/@Before and tearDown/@After options and then OK.



In the folder ‘test’ a class nl.amis.processor.SubscriptionProcessorTest.java is created.



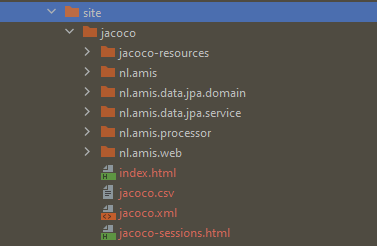
* Now run the maven verify lifecycle. This can be done from the maven menu (in the upper right corner of the IDE), or by opening a command line terminal (left menu from the bottom line) and typing the command:



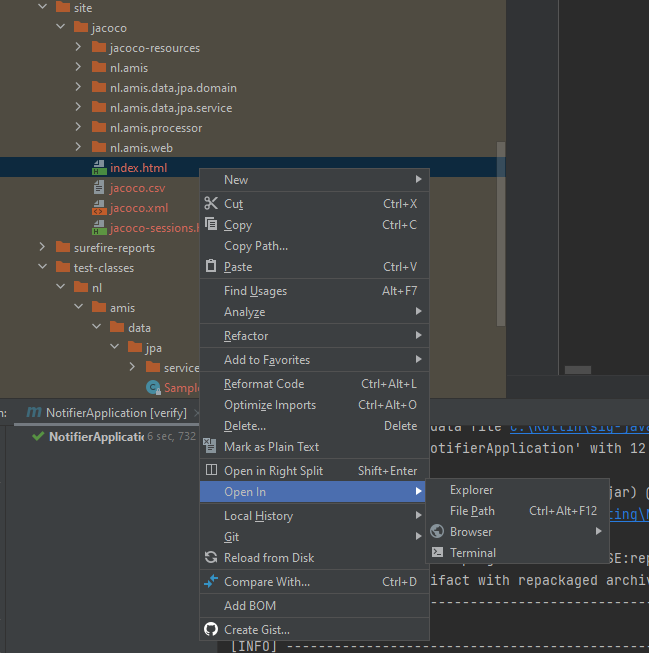


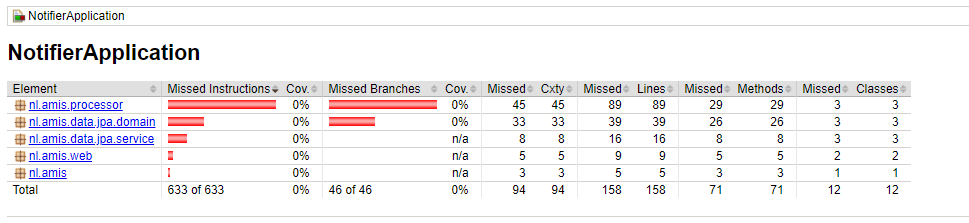
* **mvn verify**

After completion you should see the generated folder ‘target\site\jacoco’ with the file index.html



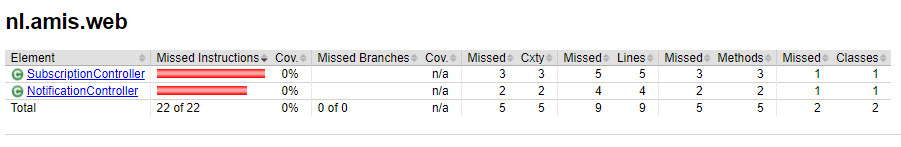
* Open the index.html file in a browser and see the ‘poor’ code coverage (right click index.html, then ‘open in’ and ‘browser’)





From the page you can drill down into the details.

Eg. Select nl.amis.web



Now let’s write some tests to improve the code coverage !

## Write some tests

### The code you will test explained

Open the file nl.amis.processor.SubscriptionProcessorTest.java (if it has been closed)



The first method you will write a test for will be getProcessorsForActiveSubscriptions() of the class file SubscriptionProcessor.java. The method has the following definition:

Map<Subscription, ChannelProcessor> getProcessorsForActiveSubscriptions() {  
  
 final Iterable<Subscription> iterable = subscriptionService.getActive(NotifierApplication.*API*);  
 final Map<String, Channel> channels = determineChannelMap(iterable);  
 final Map<Subscription, ChannelProcessor> result = createChannelProcessors(channels, iterable);  
 return result;  
}

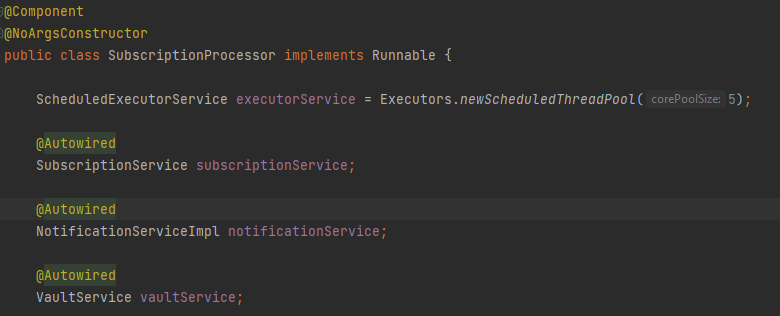
On the first line it retrieves the active subscriptions, which can be iterated over.

From these subscriptions the different channel names are determined and put in a map structure.

As a result different channel processors are created for publishing the notifications of the different subscriptions.

Looking further at the surrounding class SubscriptionProcessor.java, it can be noticed that it has some dependencies: SubscriptionService, NotificationServiceImpl en VaultService. Instances of these classes are injected as so called beans by the Spring framework during startup (because of the autowired keyword).

Also notice that this class is instantiated as a bean itself, because it is annotated with the Component keyword, and that it only has a constructor that takes no arguments



### Writing the test code

To be able to test the getProcessorsForActiveSubscriptions() method we need an instance of the surrounding class SubscriptionProcessor.java

To create this instance together with injected beans in your test class, the Mockito framework comes in handy. Mockito is able to create mocks for the SubscriptionService, NotificationServiceImpl en VaultService and inject these into another object.

* Therefor add the Mockito extension to the SubscriptionProcessorTest.java class

@ExtendWith(MockitoExtension.class)  
class SubscriptionProcessorTest {

* Also add the following global mock declarations to the SubscriptionProcessorTest.java class

@Mock  
SubscriptionService subscriptionService;  
  
@Mock  
NotificationServiceImpl notificationService;  
  
@Mock  
VaultService vaultService;

* Further add the class that contains the methods to be tested as a spy (see the below explanation between spies and mocks)

@Spy  
@InjectMocks  
SubscriptionProcessor subscriptionProcessor;

List<Subscription> iterable = Mockito.*mock*(List.class);

@Test  
void getProcessorsForActiveSubscriptions() {

The first line of code in the test method is for handling the line of code in the method getProcessorsForActiveSubscriptions of the SubscriptionProcessor.java class.

final Iterable<Subscription> iterable = subscriptionService.getActive(NotifierApplication.API);

The Notification.API has been declared of type String. The result from the above method call is an Iterable that will be walking over a collection of Subscriptions.

*The interface SubscriptionService specifies that getActive return a List, which extends Collection. Collection itself extends Iterable.*

**So, we need to instruct the code to return an Iterable containing instances of type Subscription when the method getActive(String channel) is invoked on the mock subscriptionService by our test. This is what can be done by instructing Mockito as follows:**

* **First define the data to be returned as a variable.**
* **Then instruct Mockito to return this data when the method specified is invoked on the mocked object with the specified parameter(s) (type(s)).**
* Add the following as the first 2 lines in the test method:

List<Subscription> iterable = new ArrayList<>();  
*doReturn*(iterable).when(subscriptionService).getActive(*anyString*());

If you also want to be able to check the interactions performed on the iterable, you can also use a mocked iterable instead.

* Let’s do just that by changing the added lines as follows

List<Subscription> iterable = Mockito.*mock*(List.class);  
*doReturn*(iterable).when(subscriptionService).getActive(*anyString*());

The second line of code that will be tested from the method getProcessorsForActiveSubscriptions() in SubscriptionProcessor.java class is:

final Map<String, Channel> channels = determineChannelMap(iterable);

Again, first define the data to be returned from the method invocation on the mocked object. In this case it will be a map containing channels.

Then specify that this data is to be returned, but only if the parameter on the getActive method contains some Iterable.

* Add the following lines:

Map<String, Channel> channels = Mockito.*mock*(Map.class);  
*doReturn*(channels).when(subscriptionProcessor).determineChannelMap(*any*(Iterable.class));

If you want the test only to succeed if the iterable from the first 2 lines of code is passed to the parameter (which is likely a better test), then this is also possible.

* For this you need to adjust the code as follows

Map<String, Channel> channels = Mockito.*mock*(Map.class);  
*doReturn*(channels).when(subscriptionProcessor).determineChannelMap(iterable);

The last line in the method under test is:

final Map<Subscription, ChannelProcessor> result = createChannelProcessors(channels, iterable);

The data to be returned is a map. This map must be returned when the createChannelProcessors method is invoked with the iterable and channels data.

* For this you need to adjust the code as follows

Map<Subscription, ChannelProcessor> result = Mockito.*mock*(Map.class);  
*doReturn*(result).when(subscriptionProcessor).createChannelProcessors(channels, iterable);

Now the method can be invoked for testing.

* Add this line to your code

final Map<Subscription, ChannelProcessor> outcome = subscriptionProcessor.getProcessorsForActiveSubscriptions();

### Add test assertions

After the method invocation you can add code that will verify if all things went as expected.

Let’s verify that the outcome from the getProcessorsForActiveSubscriptions() is the same as the specified result (referencing the mocked Map.class).

* For this add the following line of code:

Assertions.*assertEquals*(outcome, result);

* To verify if the getActive method has been invoked once with a String value as a parameter add:

*verify*(subscriptionService, *times*(1)).getActive(*anyString*());

* For verification if the method determineChannelMap has been invoked once with an Iterable instance as a parameter value add:

*verify*(subscriptionProcessor, *times*(1)).determineChannelMap(*any*(Iterable.class));

A better verification is if the test only succeeds if the parameter value is the iterable variable referencing the Mockito.mock(List.class) (specified in the first line of the test method).

* change the above verification to:

*verify*(subscriptionProcessor, *times*(1)).determineChannelMap(iterable);

Also add a verification for the createChannelProcessors method invocation.

* To verify if the createChannelProcessors method has been invoked with the channels and iterable mocks as a parameter value add:

*verify*(subscriptionProcessor,*times*(1)).createChannelProcessors(channels,iterable);

To be able to assert parameters used on mocked methods, you can capture the value using an ArgumentCaptor:

* Instantiate an ArgumentCaptor:

ArgumentCaptor<String> stringCapture = ArgumentCaptor.forClass(String.class);

* Replace the to-capture parameter:

*doReturn*(iterable).when(subscriptionService).getActive(stringCapture.capture());

* Add an assertion for the value:

*assertEquals*("", stringCapture.getValue());

If you’re using AssertJ, the assert can be written as:

*assertThat*(stringCapture.getValue()).isEqualTo("");

As you can see, this syntax is more natural, and may make it easier to read.

Your test method now should look as follows:

@Test  
void getProcessorsForActiveSubscriptions() {  
  
 List<Subscription> iterable = Mockito.*mock*(List.class);

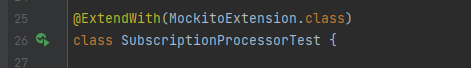
ArgumentCaptor<String> stringCapture = ArgumentCaptor.forClass(String.class);  
 *doReturn*(iterable).when(subscriptionService).getActive(stringCapture.capture());  
 Map<String, Channel> channels = Mockito.*mock*(Map.class);  
 *doReturn*(channels).when(subscriptionProcessor).determineChannelMap(iterable);  
 Map<Subscription, ChannelProcessor> result = Mockito.*mock*(Map.class);  
 *doReturn*(result).when(subscriptionProcessor).createChannelProcessors(channels, iterable);  
 final Map<Subscription, ChannelProcessor> outcome = subscriptionProcessor.getProcessorsForActiveSubscriptions();  
 Assertions.*assertEquals*(outcome, result);  
 *verify*(subscriptionService, *times*(1)).getActive(*anyString*());  
 *verify*(subscriptionProcessor, *times*(1)).determineChannelMap(iterable);  
 *verify*(subscriptionProcessor,*times*(1)).createChannelProcessors(channels,iterable);

*assertEquals*("", stringCapture.getValue());

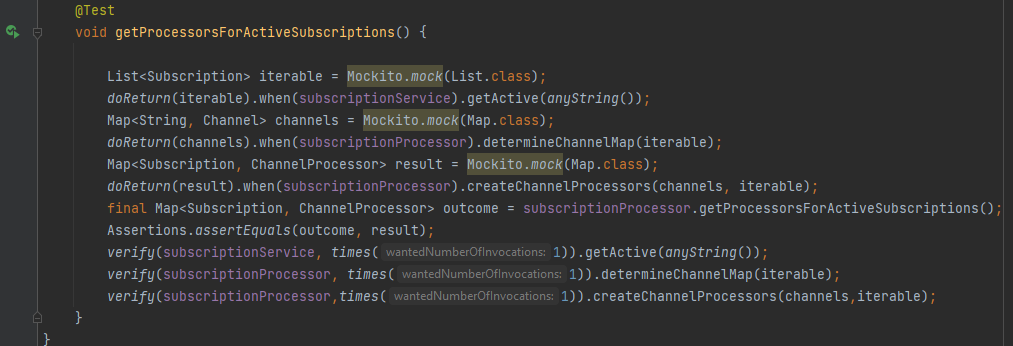
*assertThat*(stringCapture.getValue()).isEqualTo("");  
}

Now invoke the test method by pressing the run symbol at the class level which will invoke all the tests in it, or the run symbol at the method level to invoke only this, or the run option from the test runner and see if it succeeds:

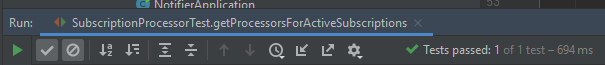
At class level:



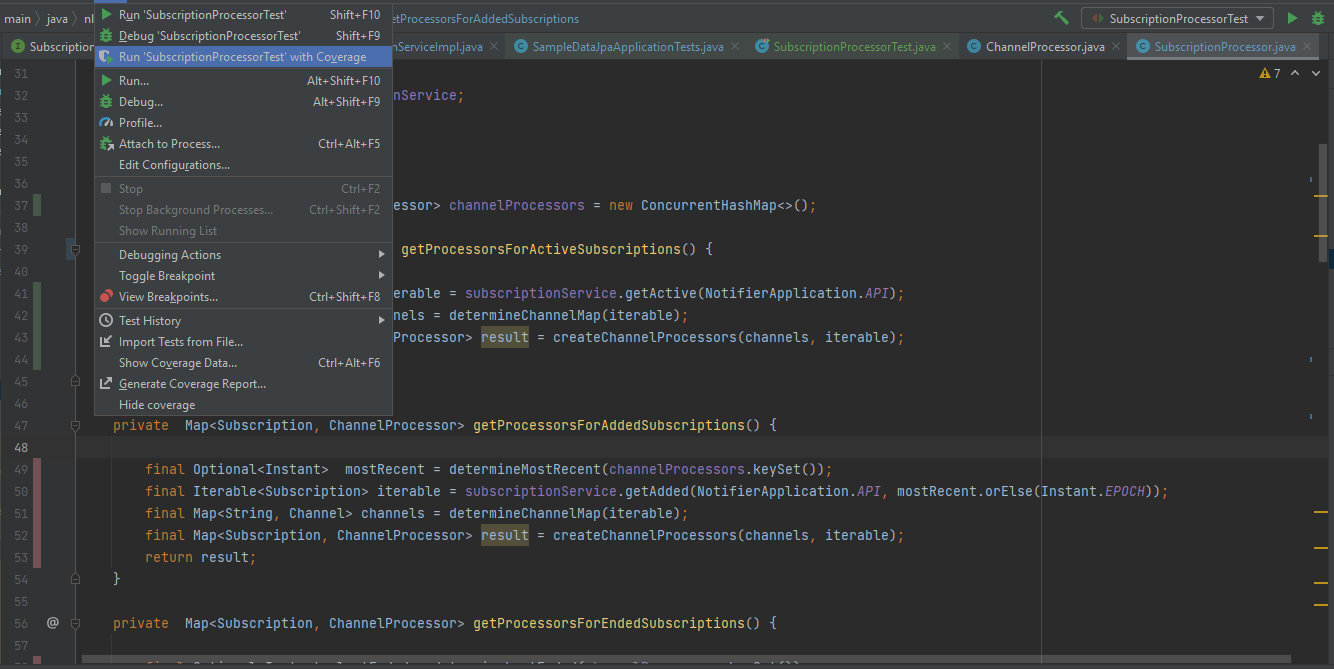
At method level:



At test runner level:

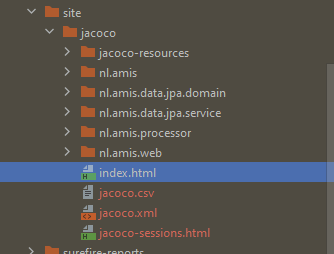


To see the coverage from within the IntelliJ IDE choose the ‘Run … with coverage option’ from the top level menu option Run. It will highlight the code on the left side of the code (as can be seen in the below screen print)

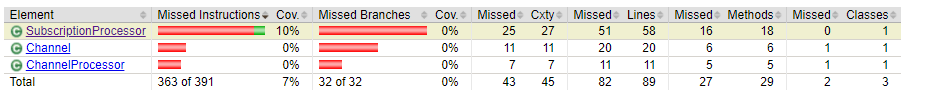


Also let’s return to the coverage report mentioned earlier.

* Before opening re-run mvn verify (if not done already) and refresh/re-open the index.html file



See that the coverage has for the SubscriptionProcessor.java has improved a little bit.



## Next exercise

You should now be able to write the tests for the other methods in this class.

Code coverage should be at a minimal of 80%!

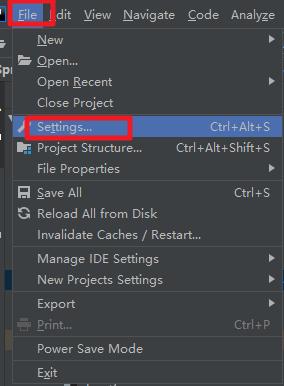
## The difference between Spy and Mock explained

On a mock annotated class you can declare the responses that must be returned for the different method invocations of it. Also the number of times a method is invoked and the different argument values for the method parameters are recorded. You can specify for one or more methods of a mock that the real method of the class must be invoked though.

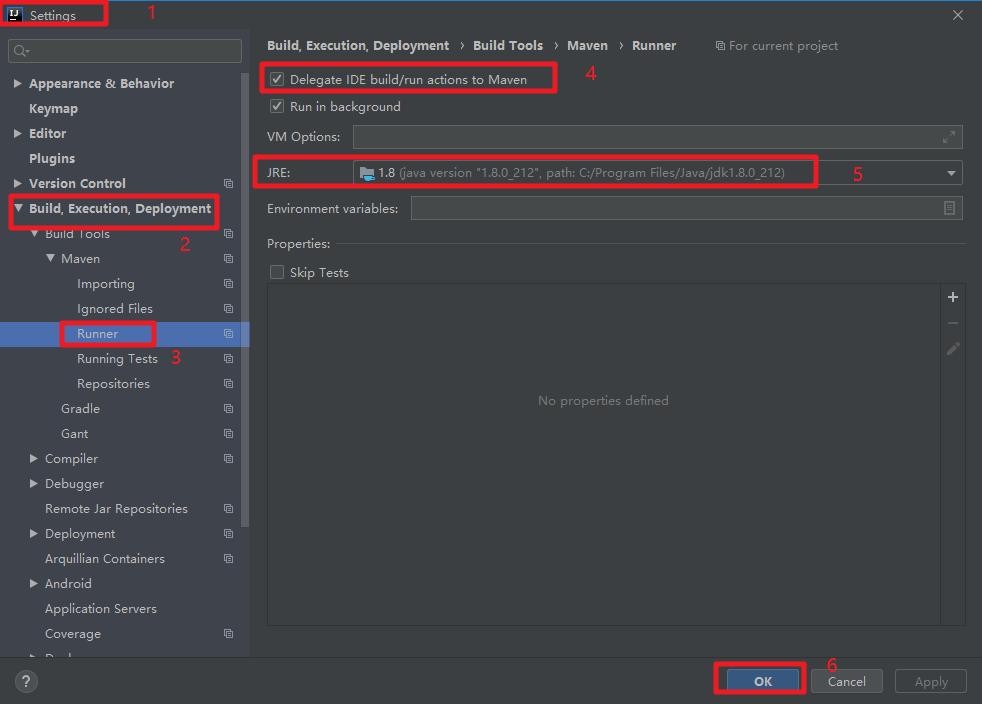
A spy annotated method measures the interactions and also records the different argument values for the method parameters, but always invokes the class its real methods.

On the problem:

**Java: package org.springframework.stereotype does not exist**



Setting-Build, Execution, Deployment-Maven-Runner-select Delegate IDE, select the path of the JRE installed by yourself-OK



Successfully started

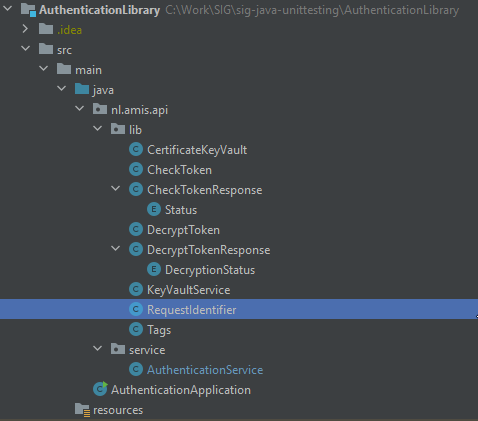
# Lab2: AuthenticationLibrary project

At <https://github.com/AMIS-Services/sig-java-unittesting> you can download a partially functional Java 11 + Spring boot application called AuthenticationLibrary.

In it you can find a service trying to collect data after decrypting the provided token, to see if the corresponding user has a valid certificate in the keyvault.

## Project structure

The structure of the NotifierApplication is as follows:



The focus of this lab is the class in nl.amis.api.service.

The AuthenticationService class has one public method, getTags(), that can be used to get the relevant data. It will use the private method getCertificate to determine the certificate associated with the parameters, and that will use the private method checkTokenAndDecrypt to perform http calls to check the token and to decrypt it.

# Pom.xml

Similar to Lab1, we need to add the correct dependencies in the pom.xml.

## JaCoCo plugin

**Code coverage** is [a software metric](https://www.baeldung.com/cs/code-coverage) used to measure how many lines of our code are executed during automated tests. For measuring and reporting on the code coverage of the tests JaCoCo will be used.

* As an initial setup in the project, add the jacoco plugin to the pom.xml file in the build section (as below).

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 <plugin>  
 <groupId>org.jacoco</groupId>  
 <artifactId>jacoco-maven-plugin</artifactId>  
 <version>0.8.4</version>  
 <executions>  
 <execution>  
 <goals>  
 <goal>prepare-agent</goal>  
 </goals>  
 </execution>  
 <execution>  
 <id>report</id>  
 <phase>prepare-package</phase>  
 <goals>  
 <goal>report</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

## Setup Junit 5 dependencies

* First exclude Junit 4 from the spring-boot starter-test dependency in the pom.xml as follows:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 <exclusions>  
 <exclusion>  
 <groupId>org.junit.vintage</groupId>  
 <artifactId>junit-vintage-engine</artifactId>  
 </exclusion>  
 </exclusions>  
</dependency>

* Then add the dependency for Junit 5

<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter</artifactId>  
 <version>5.7.2</version>  
 <scope>test</scope>  
</dependency>

# Writing Junit tests

You perform the same steps as in Lab 1 to create the test classes and prepare the AuthenticationService to be tested.

To summarize:

* Generate a test class;
* Make the to be tested methods package visible;
* Generate a test for each method;
* Run mvn verify.

## Write some tests

### Writing the test code

To be able to test the getTags() method we need an instance of the surrounding class AuthenticationService.java

To create this instance together with injected beans in your test class, the Mockito framework comes in handy. Mockito is able to create mocks for the SubscriptionService, NotificationServiceImpl en VaultService and inject these into another object.

* Therefor add the Mockito extension to the AuthenticationServiceTest.java class

@ExtendWith(MockitoExtension.class)  
class AuthenticationServiceTest {

* Also add the following global mock declarations to the SubscriptionProcessorTest.java class

@Mock  
WebServiceTemplate webserviceTemplate;  
  
@Mock  
KeyVaultService keyVaultService;  
  
@Mock  
TelemetryClient telemetryClient;

* Further add the class that contains the methods to be tested as a spy (see the below explanation between spies and mocks)

@Spy  
@InjectMocks  
AuthenticationService authenticationService;

@Test  
void testGetTags() {

You can see that the parameter used is from the RequestIdentifier class. This class doesn’t have a constructor, so you cannot initialize it. The public method available is create().

This will create an RequestIdentifier object from the query parameters of the current request. Since we’re not invoking an actual http call, we also have to mock this.

This can be done with the following code:

String query = "oid=999&goid=888&tkn=token&apk=APK";  
MockHttpServletRequest request = new MockHttpServletRequest();  
request.setQueryString(query);  
RequestContextHolder.*setRequestAttributes*(new ServletRequestAttributes(request));

Then you can create a RequestIdentifier to use as parameter:

RequestIdentificer requestIdentifier = RequestIdentifier.*create*();

Then, we need to instruct the code to return mocked instances of specific classes when the marshalSendAndReceive() is invoked, again:

* **First define the data to be returned as a variable.**
* **Then instruct Mockito to return this data when the method specified is invoked on the mocked object with the specified parameter(s) (type(s)).**
* Add the following lines in the test method:

CheckTokenResponse checkResponse = new CheckTokenResponse();  
checkResponse.setStatus(CheckTokenResponse.Status.VALID);

*doReturn*(checkResponse).when(webserviceTemplate).marshalSendAndReceive(*any*(CheckToken. class), *any*(SoapActionCallback.class);

As you can see, we now define a specific class to be the expected parameter instead of just any().

* Add similar lines of codes for the DecryptTokenResponse, giving it a mocked Username to return:

DecryptTokenResponse decryptResponse = new DecryptTokenResponse ();  
decryptResponse.setDecrypted("Username");

*doReturn*(decryptResponse).when(webserviceTemplate).marshalSendAndReceive(*any*(DecryptToken. class), *any*(SoapActionCallback.class);

Here you can see why we use the specific class instead of any(). Since both checking and decrypting a token uses the same method, marshalSendAndReceive(), you can specify the parameters to distinguish both calls.

When you know what the order is, you could also replace both doReturn() lines with:

when(webserviceTemplate).marshalSendAndReceive(*any*(Object.class), *any*()).*thenReturn*(checkResponse).*thenReturn*(decryptResponse);

The any(Object.class) is important, because it’s an overloaded method with multiple 2 parameter options.

* Also add the mock return for the KeyVaultService:

Tags mockTags = new Tags(“APK”, “Username2”, “RoleGroup”);  
CertificateKeyVault certificate = new CertificateKeyVault (“certificate”, “value”.getBytes(StandardCharsets.UTF\_8), “thumb”, null, null, mockTags, null);

*doReturn*(certificate).when(keyVaultService).getCertificateByUsername(*anyString*());

Now the method can be invoked for testing.

* Add this line to your code

final Tags tags = authenticationService.getTags(requestIdentifier);

### Add test assertions

After the method invocation you can add code that will verify if all things went as expected.

Let’s verify that the outcome from the getTags() is the same as the specified result (referencing the mocked Tags.class).

* For this add the following line of code:

Assertions.*assertEquals*(tags.getUsername(), mockTags.getUsername());

* To verify if the marshallSendAndReceive method has been invoked twice with an Object.class as a parameter add:

*verify*(webserviceTemplate, *times*(2)). marshalSendAndReceive(*any*(Object.class), *any*()));

* For verification if the method getCertificateByUsername has been invoked once with a String as a parameter value add:

*verify*(keyVaultService, *times*(1)).getCertificateByUsername(*anyString*());

To be able to assert parameters used on mocked methods, you can capture the value using an ArgumentCaptor:

* Instantiate an ArgumentCaptor:

ArgumentCaptor<String> stringCapture = ArgumentCaptor.forClass(String.class);

* Replace the to-capture parameter:

*doReturn*(iterable).when(subscriptionService).getActive(stringCapture.capture());

* Add an assertion for the value:

*assertEquals*("", stringCapture.getValue());

If you’re using AssertJ, the assert can be written as:

*assertThat*(stringCapture.getValue()).isEqualTo("");

As you can see, this syntax is more natural, and may make it easier to read.

Your test method now should look as follows:

@Test  
void testGetTags() {  
  
 String query = "oid=999&goid=888&tkn=token&apk=APK";

MockHttpServletRequest request = new MockHttpServletRequest();

Request.setQueryString(query);

RequestContextHolder.setRequestAttributes(new ServletRequestAttributes(request));

RequestIdentificer requestIdentifier = RequestIdentifier.*create*();

CheckTokenResponse checkResponse = new CheckTokenResponse();  
 checkResponse.setStatus(CheckTokenResponse.Status.VALID);

*doReturn*(checkResponse).when(webserviceTemplate).marshalSendAndReceive(*any*(CheckToken. class), *any*(SoapActionCallback.class);

DecryptTokenResponse decryptResponse = new DecryptTokenResponse ();  
 decryptResponse.setDecrypted("Username");

*doReturn*(decryptResponse).when(webserviceTemplate).marshalSendAndReceive(*any*(DecryptToken.class), *any*(SoapActionCallback.class);

Tags mockTags = new Tags(“APK”, “Username2”, “RoleGroup”);  
 CertificateKeyVault certificate = new CertificateKeyVault (“certificate”, “value”.getBytes(StandardCharsets.UTF\_8), “thumb”, null, null, mockTags, null);

*doReturn*(certificate).when(keyVaultService).getCertificateByUsername(*anyString*());

final Tags tags = authenticationService.getTags(requestIdentifier);

Assertions.*assertEquals*(tags.getUsername(), mockTags.getUsername());

*verify*(webserviceTemplate, *times*(2)). marshalSendAndReceive(*any*(Object.class), *any*()));

*verify*(keyVaultService, *times*(1)).getCertificateByUsername(*anyString*());

}

Now invoke the test method by pressing the run symbol at the class level which will invoke all the tests in it, or the run symbol at the method level to invoke only this, or the run option from the test runner and see if it succeeds:

Also let’s return to the coverage report mentioned earlier.

* Before opening re-run mvn verify (if not done already) and refresh/re-open the index.html file