UDACITY

PROJECT SPECIFICATION

UdaSecurity

Code Functionality

CRITERIA	MEETS SPECIFICATIONS
Write code that runs without errors	You should submit a project that can compile and run without errors. After making all of the changes required for this project, you should still be able to launch the application, see the application GUI, and perform its operations without exceptions.
Fix the application to correctly perform all the behaviors in the requirements list	You will use your unit tests to find any missing or improperly implemented application requirements. Based on your unit test findings, make sure the application correctly implements all requirements.
Correct any High priority errors found by SpotBugs in your build report.	After you've finished the building and compiling objectives, you should be able to view a list of bugs found by the SpotBugs plugin. If any of the bugs are `High` priority, you should fix them before submitting your project.

CRITERIA	MEETS SPECIFICATIONS
(Optional) Switch image service implementation to use the AwsImageService	Modify the dependencies injected in the main GUI class to create an instance of the AwsImageService instead of the FakeImageService. For this AwsImageService to work, you will need to create an IAM user with AWS Rekognition access and create a config.properties file in the `src/main/resources` directory of the ImageService module so the AwsImageService can connect. More instructions are in the AwsImageService.java file.

Building and Compiling

CRITERIA	MEETS SPECIFICATIONS
Add dependencies in `pom.xml` file	Your starter project will be missing Maven dependencies. You must identify all missing dependencies and add them to the `pom.xml` file.

CRITERIA	MEETS SPECIFICATIONS
Update `pom.xml` after separating project into module to move shared dependencies into the parent `pom.xml`	The parent pom should not contain dependencies that are required by only one of the services. The same dependencies should not be found in multiple `pom.xml` files. You should be able to build the project by running the `mvn compile` command from the project parent directly. You should also be able to run the application main class from within IntelliJ.
Build project into executable JAR	After separating the project into modules, you will need to build the project into an executable JAR. You should do this using a Maven Plugin. Submit a screenshot titled "executable_jar.png" that shows you running the executable jar from the command line and the application launching. Use the command `java -jar [yourjarname]` to run it.

CRITERIA	MEETS SPECIFICATIONS

Configure the `pluginmanagement` section of the pom.xml to support our unit test libraries from maven.	After separating your project into modules and writing your unit tests, you want to confirm that you can run the unit tests by executing the `mvn test` phase from the parent project directory.
	
	To do this, you will need to configure the maven-surefire-plugin to use the latest version and add a configuration element that specifies an argument that opens the test package of your module to all unnamed modules, so that Mockito can access your objects.
	
	For example:
	
	`<configuration>`
	` <argLine>`
	`   --add-opens [modulename]/[packagename]=ALL-UNNAMED`
	` </argLine>`
	`</configuration>`
	
	To complete this objective, you must run these tests by running the command `mvn test` from the directory containing the parent pom of your multi-module project and verify that your unit tests execute.

CRITERIA	MEETS SPECIFICATIONS
Configure the `reporting` section of the pom to run the spotbugs plugin.	Create a `reporting` tag in the pom if it's not already there and then add the spotbugs plugin. You should be able to run this plugin as part of your maven build process using the `mvn install site` command. If this executes successfully, you should find a file called spotbugs.html in the /target/site/ directory of your security module, which will identify any potential bugs found by the static analysis tool. You should include this spotbugs.html as part of your project submission.

Modules

CRITERIA	MEETS SPECIFICATIONS
Create a parent project with two modules: one for `ImageService` and one for `SecurityService`	Both the ImageService and SecurityService should have their own module in the project structure. One way to do this is to create a parent project that contains two separate modules. You should separate the files in the starter project so that they are contained in the appropriate modules. The ImageService module should only contain classes specific to the image analysis, and everything else can go in the SecurityService module.

CRITERIA	MEETS SPECIFICATIONS
Create a module descriptor for each module	Each service's module descriptor should export any packages required by the other service and they should require the package from the other service that they need. Try to limit the scope of dependencies exchanged. Your image service should not require any packages from the security service, for example.
Add dependencies and transitive dependencies to module descriptors	Many of the maven dependencies used by this project will require your modules to specify which packages are required by the project, and so you must add `requires` statements to make those libraries available to your project. IntelliJ should be able to help you identify these missing `requires` statements. There may also be cases in which transitive dependencies are not properly declared by your source libraries, and so you may need to add `requires` statements to make these libraries available. A missing transitive dependency will usually produce a `ClassNotFoundException` when building, which you can resolve by `requires [package containing the missing class]`. Lastly, you will need to open at least one package to reflection, because our fake repository implementation uses the Gson library to perform serialization through reflection. Open only the package necessary and limit the scope of your open package to the library that needs it.

Unit Tests

CRITERIA	MEETS SPECIFICATIONS
Create a unit test file for the Security Service	You should create a class that contains unit tests for the SecurityService. This class should be called SecurityServiceTest. Make sure it's created in the correct scope and correct folder to test the Security Service. Use the maven directory structure convention for your tests directory and make sure the package name matches the package containing SecurityService.
Write unit tests that test all application requirements are properly implemented by the SecurityService class.	All of the requirements that are listed in the instructions should be tested. Our goal is specifically to test the SecurityService class, and so you should avoid writing unit tests that rely on the behavior of the Repository or ImageService. Each of the application requirements in the instructions should have at least one unit test that verifies that the application will perform this behavior. Some of them may require two or more tests to appropriately confirm the behavior of multiple scenarios or failure conditions.

CRITERIA	MEETS SPECIFICATIONS
Write unit tests that provide coverage of all substantial methods in the SecurityService class	You must have full coverage of methods that implement the application requirements. You don't need to write unit tests for trivial methods like getters or setters (e.g. `getAlarmStatus`), but IntelliJ code coverage should demonstrate that all methods with more than 1 line of code are tested by your unit tests. Furthermore, all the lines in these methods should be reachable by your tests, so make sure your unit tests explore all branching conditions.

CRITERIA	MEETS SPECIFICATIONS
Provide constructive unit test names	Make sure that your unit test names communicate: • What functionality the test is assessing • The condition under test or the inputs to the test • The expected output You should avoid using the word ‘test’ in your names as well, because this is redundant and you can use that space to communicate what the test does. Good test name examples: • `sensorActivated_alarmArmedAndStatusPending_alarmStatusAlarm` • `imageAnalyzed_catDetectedAndStatusArmedHome_alarmStatusAlarm` Bad test name examples: • `testSensorActivated` • `requirement3`

CRITERIA	MEETS SPECIFICATIONS
Utilize the features of JUnit to simplify your unit tests	At least one unit test should use **Parameterized**, and probably more than one. Any time you can identify multiple similar tests with slightly different input conditions, consider using Parameterized or Repeated tests to reduce the duplication. If you need to prepare commonly-used tested objects for each unit test, use the `@BeforeEach` annotation to reduce duplicated code. This is a great place to construct the SecurityService instance under test, as well as initialize any other commonly used variables.
Test only the SecurityService	Your units tests should only test the Security Service. Use the features of Mockito to mock the dependencies of SecurityService so that your unit tests do not rely on the ImageService or SecurityRepository functioning correctly. If certain functionality that you need to test is in other services or in the GUI, you will need to refactor your code to move the code that implements that functionality into the Security Service. Remember, if the image service or repository are broken, this should not break your unit tests as they only test the Security Service. You should create `@Mock` image service and repository using Mockito so that you can control their outputs in your unit tests.

CRITERIA	MEETS SPECIFICATIONS
(Optional) Create integration tests	Sometimes you may wish to test end-to-end functionality. However, our repository implementation doesn't lend itself well toward testing, because it writes to local properties every time it is used. Create a FakeSecurityRepository class that works just like the PretendDatabaseSecurityRepository class except without the property files. Create a second test class called `SecurityServiceIntegrationTest.java` and write methods that test our requirements as integration tests. These tests can call service methods and then use JUnit Assertions to verify that the values you retrieve after performing operations are the expected values.

Suggestions to Make Your Project Stand Out!

1) Write a unit test that uses a **spy**.

2) Write integration tests that tests the interaction between multiple services. (i.e. Write a test that confirms if you change the UI that the database gets updated correctly.) *Note:* Create a "fake" repository instance so you can test the interaction between the Service class and a repository.

3) Replace the FakeImageService with the AwsImageService so you can actually scan images for cats!