VPN

In this exercise you'll build a scheduled service that monitors a simulated network-status indicator. Your application will check the status every few seconds, and when it changes you'll publish a message to a JMS topic with the latest status value.

You'll build the project just about from scratch: there is a starter Maven project with a POM, a starter data file **Status.txt**, and a variant of our **Subscriber** class to help with testing. Do all of your work in a new package **com.amica.network**.

1. Create an application Spring Boot application class **VPN**. Annotate it to enable Spring scheduling. Give it the usual **main** method to **run** the application.
2. Create a Spring component class **Monitor**. Give it a scheduled method and indicate that Spring should call the method every 5 seconds.
3. Define an enumerated type **Status** with values **RUNNING, STOPPED, and STARTING.**
4. In the method read the contents of the text file **Status.txt**. Attempt to parse the contents, – which should be a single line with a single word on it – as one of the **Status** values. Catch any exceptions encountered in this conversion attempt, and log a warning that the file contents were not a recognized value. (You might also log a warning if the file is not found.)
5. Log the status at INFO level.
6. Add configuration-manager folders to **src/main/resources**, with any component name you like and an environment name of “developerworkstation”. All folders can hold dummy files for the moment, but you'll need the folders and files to be there since the **component-configuration-manager-spring-boot** dependency will auto-configure and come looking for them.
7. Add a line to **VPN.main** to set the **env.name** property to “developerworkstation”.
8. Test by running your application, and you should see the log messages. Change the contents of the file, while the application is running, and see that your component picks up on the changes.

9. Start keeping track of the status over multiple calls, and detect changes. This will require a field, something like **latestStatus**, on the class, where you'll store status after reading it and then you'll compare the value you read from the file the next time around. For now, just print a message when you detect a change. Test this by running the application.

10. Add a **@Value**-injected field **filename** and use this instead of the hard-coded "Status.txt". Use any property name you like. Add the **@Setter** annotation, too, so you can set this value up in a unit test.

11. Set up a properties file in the environment resources that defines the filename, so your component will get it at runtime. Test by running the application.

12. Get a unit test going: create a **Monitor** object, set its filename programmatically to a separate file, such as **TestStatus.txt**, and call its scheduled method directly. You can't really assert anything useful just yet, but you should see the same log output.

13. Create a helper method **setStatus** that takes a **Status** value and writes it to your test status file, overwriting whatever was there. Then try changing the contents of the file between calls to your scheduled method, and see that it picks up the changes.

14. Get an integration test going: make this a Spring test, and use the **VPN** class as your configuration, but define a **@BeforeAll** method that sets **env.name** to "Connected" instead of to "developerworkstation".

15. Set up an alternate environment-resources folder structure for this environment namem under **src/test/resources**. Set the status filename to your test file, so that neither your unit nor integration tests will ever disrupt the working application.

16. Copy in your **setStatus** helper, and do similar tests – except that, instead of calling the scheduled method directly, you'll sleep for 5 seconds after making a change to the status file, and in that interval you should see the updated status in the logging output.

    Also, it's going to be best to stick with a single test method in this class. The scheduled calls to the test target, along with a shared Spring context for any additional test methods, is going to make it quite unwieldy to set up and then test for status changes in multiple methods – especially given that they will not run in a predictable order. So define a single method and run through a simple scenario of status changes.

17. Get the **JMSConfiguration** class from one of our JMS exercises, and make a copy in the **com.amica.network** package. Change the integration qualifier to "VPN". Do not include the **@EnableJms** annotation, since we'll be publishing messages but not listening for them and so we don't need a long-running listener container.
18. **@Import** this configuration into your application class.
19. Give your **Monitor** component an autowired JMS template, and a **@Value**-injected string that will be the name of the JMS topic.
20. In your scheduled method, use the template to **convertAndSend** the string representation of the status, only when the status has just changed.
21. Now you're in a position to test the component behavior in a more assertive way. In your unit test, use a mock JMS template, a programmatically set topic name, and Mockito's **verify** capability to prove that the component publishes the expected message when the status changes. Remember, there should only be an outbound message if the status has changed, so test for both cases where it does and cases where it doesn't.
22. The integration test will be more involved, because you'll need to subscribe to the topic in order to see the published message. Start by setting up JMSSpring properties for your "VPN" qualifier, in a file **MQ.properties** placed in your "Connected" environment folder. Be sure to set the pub/sub domain value to true, so that the component will publish to a topic rather than sending to a queue. Transactional and subscription-durable values can be **false**. Also set your topic-name property to "TestTopic".
23. Notice that the **Subscriber** class is defined to be available only in the "test" profile. This makes it a little simpler to include it in the integration-test configuration while keeping it out of the application. Add the **@ActiveProfiles** annotation to your integration-test class, specifying the "test" profile, and add an autowired field **subscriber**.
24. The **Subscriber** simply records all the messages it receives from the topic, and you can **reset** it at any time for a clean slate. So now you can test the running component by changing the test status file, one time or a few times, with the same value or different values; wait 5 seconds between changes; and inspect the received message by calling **subscriber.getReceived** to see that the right notifications are being published at the right times.

    One other tip here: sleep for 5 seconds <u>after</u> your last status change, to give the component time to pick up on it before you start asserting that messages were received by the subscriber.

25. When both unit and integration tests are working well, configure the JMS properties under **src/main/resources/EnvironmentResources/developerworkstation**, in an **MQ.properties** file. All of the JMSSpring properties are the same as for the integration test, but set the topic name to "VPNStatus".

26. Now, you can run the application, but you won't really be able to tell if it's working, except that it doesn't crash. You'll see the log output, but the published messages will just go on a topic and sit there, since no one's subscribing …

27. Time for a functional test! You're going to be mostly on your own for this one. But notice that there's not a ton of difference between the integration test and the functional test, so you can borrow a lot of code or even use the integration test as a template.

    Major differences include:

    - The integration test acts on the **Monitor** component, in a prepared Spring context that enables scheduling. For the functional test, you'll already have the **VPN** application running.
    - The integration test can set the stage for a series of status changes, basically working from scratch in the test status text file. The functional test doesn't know what might have transpired before it starts, so for example it won't be able to set a "starting" status but will have to treat its first write to the "real" status text file as possibly a change and possibly not a change. Simply **reset**ting the **subscriber** after that first change is the simplest way to get that clean slate that we test-writers crave.

28. If you have time, you might refine the integration and scheduled tests to prove out the timing of the JMS messaging, testing the "JMSTimestamp" header of the messages as they are observed by the subscriber component. Of course you can't assert exact values, but you can assert that the timestamp for a given message falls within a range of a few seconds.