Billing8-9 Implementation

Starter code: Billing8

Broadly speaking, the new application will take the following shape:

- A new **InvoicePayer** can be a **@Component**, with constructor-injected dependencies on **Updater** and **Reporter**.
- The **Reporter** and **Updater** will become Spring **@Component**s. They'll still use the configuration manager to find the data files; but they'll use an **@Autowired** dependency to get their parser or producer.
- The need to get filenames from the Amica configuration and the parser from Spring dependency injection will change the lifecycle of both of these components: the **readData** and **load** methods will no longer be called from the constructors, since it would then be called before the necessary parser had been configured. Instead you'll annotate these methods with **@PostConstruct**, so Spring will call them at the right time, automatically.
- An application **InvoicePayment** will set up a Spring application context, including a **@Bean** method that internally calls the **ParserFactory**. That class will continue to use the configuration manager to find the right parser class.
- Most tests will be only slightly affected. **ReporterTest**, **ReporterIntegrationTest**, and **UpdaterIntegrationTest** will adjust to changes in constructor signatures, and they'll need to set the target class' parser or producer explicitly, and also call **readData** or **load** explicitly, simulating what Spring would do in an application context.
- **ReporterConfigurationTest** and a new **InvoicePayerIntegrationTest** will be Spring tests, tapping into the **InvoicePayment** configuration as their context configurations.
- A new **InvoicePayerTest** will use mocks to confirm the behavior of the class under test, without needing to create a Spring test context.

As a team, you'll migrate the application to Spring more or less in the order of tasks shown below, though some of these can be done in parallel:

Create the Spring Application

1. Add dependencies to the POM for Spring: the necessary dependencies are provided in **Dependencies.txt** and should be added to **pom.xml** after the Jackson dependency and before the configuration manager.
2. You can remove the three test-scope dependencies – JUnit, Hamcrest, and Mockito – as these are all transitive dependencies of **spring-boot-starter-test**.

Create the Spring Application

3. Set up the **InvoicePayment** application first, because this will give you an easy way to check on the viability of your components and their dependencies as they come online in later steps.
4. Do not make this class a **@SpringBootApplication**, because it will otherwise try to auto-configure a few things that will just get in your way. Just make it a **@Configuration** and set it to **@ComponentScan**.
5. Give it a **@Bean** method that calls the **ParserFactory** to create a default parser.
6. Give it a **main** method, call **SpringApplication.run** and pass the class as the configuration. Use this configuration to get beans of various types as you develop, starting with the **Producer** that you're publishing with your **@Bean** method: call **getBean** on the application context returned from the **run** method, passing **Producer.class**, and print out the class name of the object or otherwise prove that you got it.
7. Set up appropriate folders and property files to support the configuration manager, and run the application with **server.env** set to "developerworkstation". Use any of the supported formats, and be sure to configure filenames that are not in your **src** tree or otherwise under source control – some local folder that you can each create, manage, and clean up independently is best.

When you have that much working, push so that teammates can use it as a sort of test bench as they take on some of the next steps.

Refactor Updater

8. Refactor **Updater** so that it no longer takes a **Parser.Format** nor tries to initialize the **parser** field in any of its constructors.
9. Instead, annotate the **parser** field as **@Autowired**, and also give it Lombok **@Getter** and **@Setter** annotations, which will support non-Spring tests.
10. Also, stop calling **load** from the constructors. Instead, annotate it as a **@PostConstruct** method so that Spring will call it once all of the component configuration is complete.
11. Now make the class a **@Component**, and you should be able to call **getBean** on the context in **InvoicePayment.main** and get a pre-configured **Updater**, with an injected **Producer** that lets it load data when it's created and configured.
12. Adjust **UpdaterIntegrationTest** to the new constructor signatures, and be sure to call **setParser** and **load** explicitly after creating each test **Updater**, since it no longer happens in the constructor and Spring isn't doing it for you in this test.

Refactor Reporter

These steps are similar to those for the **Updater** component, but there's a little more to do, because **Reporter** supports a constructor that takes **Reader**s as well as those that derive file names, and those two paths need to be channeled into one reading process, even now that we're taking an autowired parser ...

13. Refactor **Reporter** so that it no longer takes a **Parser.Format** nor tries to initialize the **parser** field in any of its constructors.
14. Instead, create a **parser** field, like the one in **Updater**, and annotate it as **@Autowired**, and also give it Lombok **@Getter** and **@Setter** annotations.
15. Also, stop calling **readData** from the constructors.
16. Instead, add two fields to the class, **customerReaderFactory** and **invoiceReaderFactory**, both of type **Supplier<Reader>**. Each constructor can now set these fields appropriately: those that were creating **FileReader**s can set the fields to lambda expressions that do the same thing on demand (though this will take some extra work to deal with the checked **IOException** inside the lambda), and those that already have readers can create lambdas that just return the readers as they already exist.
17. Make **readData public**, annotate it as a **@PostConstruct** method, and remove all of its parameters. Instead, add a resources section to the existing **try** block in which you create two **Reader**s by calling **get** on each of the two factory fields. This will let the existing parsing logic roll ahead, and will assure that any file readers are appropriately closed afterwards.
18. Now make the class a **@Component**, and you should be able to call **getBean** on the context in **InvoicePayment.main** and get a pre-configured **Reporter**, with an injected **Producer** that lets it load data when it's created and configured.
19. Update the three tests for **Reporter**, adjusting to the new constructor signatures and lifecycle. Two of the tests will need to call **setParser** and **readData** themselves. **ReporterConfigurationTest** can be promoted to a Spring test by adding **@RunWith** and **@ContextConfiguration** annotations. Use **InvoicePayment.class** as the context configuration class. It can then autowire a reference to a pre-configured **Reporter** and test as it already does.

Create the InvoicePayer Component

20. Create a new class **InvoicePayer**. Give it fields of type **Updater** and **Reporter**, and give it a constructor that initializes both.
21. Create a method **pay** that takes any number of integers (use varargs, an array, a list, whatever you like) and calls **payInvoice** on the **Updater** for each of them. Handle any exceptions (bad invoice number, invoice already paid) by logging a warning and continuing to try other numbers. Remember to **save** when done!
22. Then return the results of a call to **getOverdueInvoices** on the **Reporter**.
23. Make the class a **@Component**.
24. Now **InvoicePayment** will start to work as a practical application. Remove any tests you've done of configurations of other beans, but keep your call to **SpringApplication.run** and use the returned context to get a bean of type **InvoicePayer**.
25. Translate all command-line arguments to invoice numbers, and pass the array or list of them to the **pay** method on the component. Print out some simple representation of the overdue invoices returned from the call.
26. Test by running with a number or two in the Program Arguments section of your Eclipse run configuration.

    You may well hit an issue here: even if everything else works, and you can see that your paid invoices are "sticking" in the underlying invoices data file, the reported overdue invoices won't appear to be any different. This has to do with the lifecycle of Spring singleton beans: by default they are eagerly instantiated when the context is created, and that means that the **Reporter** has already read the data files by the time the **Updater** makes any changes.

    There are a few ways you might address this issue. You could take away the **@PostConstruct** annotation from **readData**, and then the **InvoicePayer** and all tests would have to call the method explicitly, at the appropriate time. Or, you could configure the component to be a prototype bean, and then have the **InvoicePayer** use an injected **ApplicationContext** to get a new instance of the bean only after it had done its updates.

27. Create an integration test and a unit test for the new component.