# Structuring and Testing Modules and Layers with Spring Boot

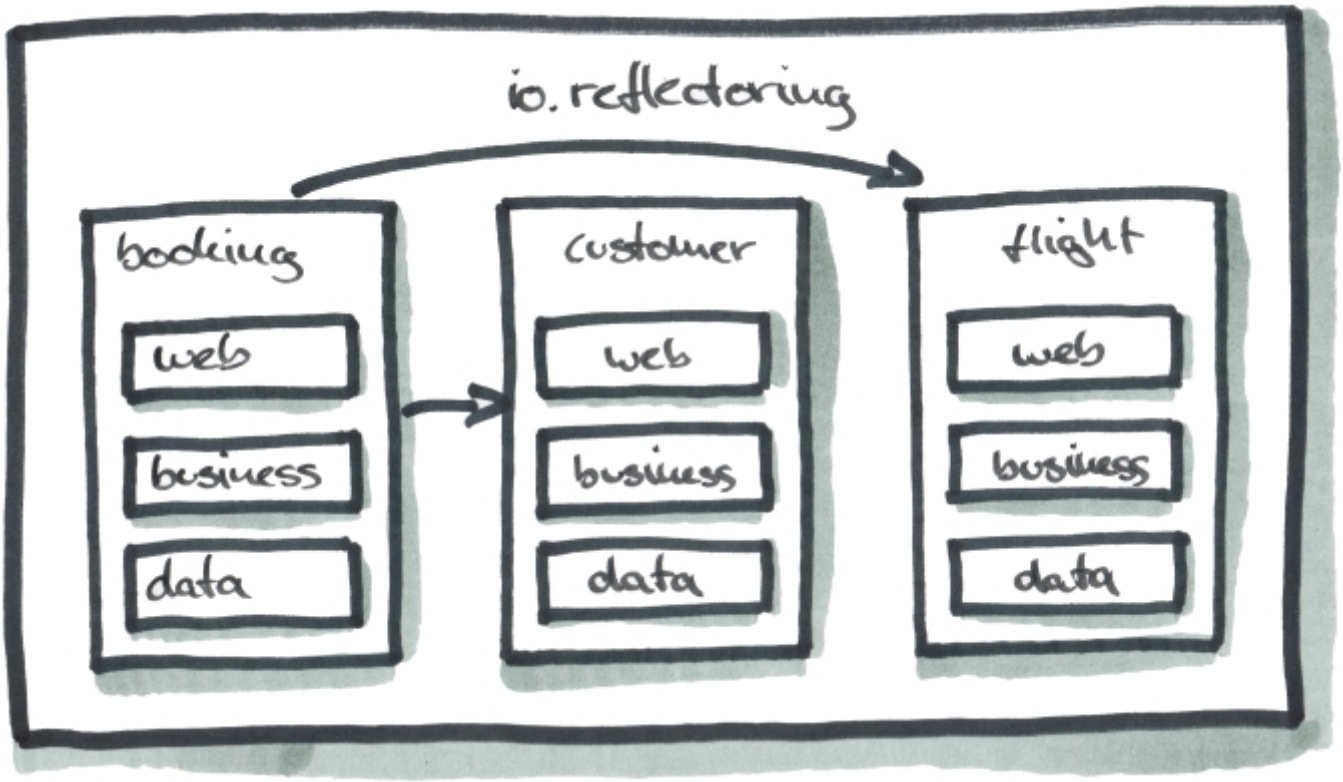
8 minute read (1704 words)

Well-behaved software consists of highly cohesive modules that are loosely coupled to other modules. Each module takes care from user input on the web layer down to writing into and reading from the database. **This article presents a way to structure a Spring Boot application in vertical modules and horizontal layers and shows ways how to test the verticals and layers with the testing features provided by Spring Boot**.

**Code Example**
This article is accompanied by working example code on github (https://github.com/thombergs/code-examples/tree/master/spring-boot/spring-boot-testing).

# Code Structure

Before we can test modules and layers, we need to create them. So, let's have a look at how the code is structured. If you want to view the code while reading, have a look at the github repository (https://github.com/thombergs/code-examples/tree/master/spring-boot/spring-boot-testing) with the example code.




The application resides in the package `io.reflectoring` and consists of three vertical modules:

- The `booking` module is the main module. It provides functionality to book a flight for a certain customer and depends on the other modules.
- The `customer` module is all about managing customer data.
- The `flight` module is all about managing available flights.

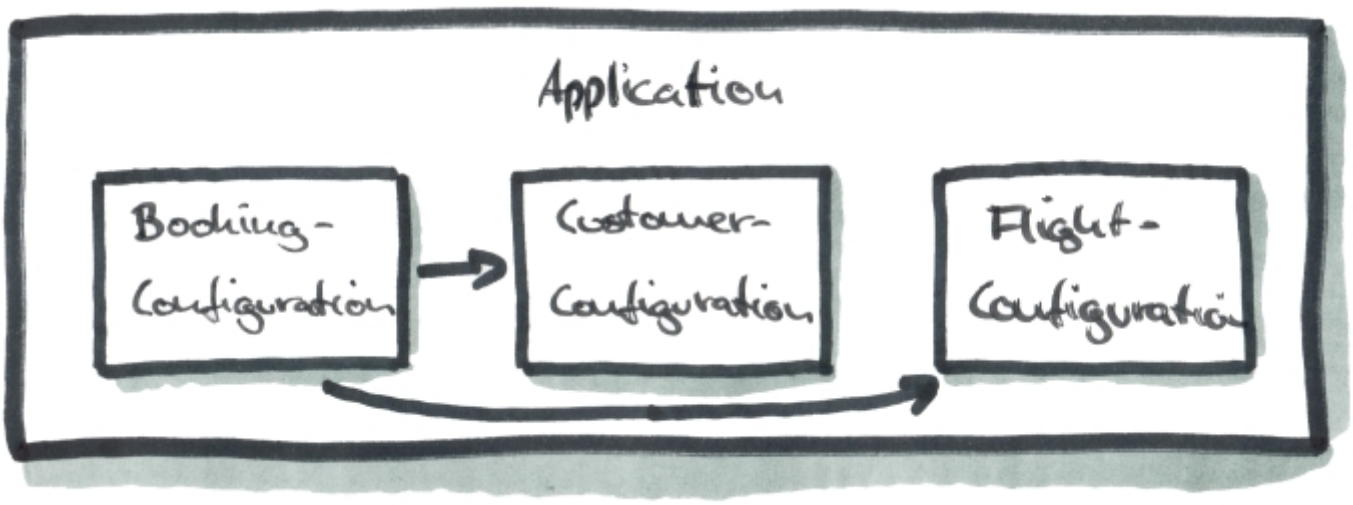
Each module has its own sub-package. Within each module we have the following layers:

- The `web` layer contains our Spring Web MVC Controllers, resource classes and any configuration necessary to enable web access to the module.
- The `business` layer contains the business logic and workflows that make up the functionality of the module.
- The `data` layer contains our JPA entities and Spring Data repositories.

Again, each layer has its own sub-package.

# `ApplicationContext` Structure

Now that we have a clear-cut package structure, let's look at how we structure the Spring `ApplicationContext` in order to represent our modules:




It all starts with a Spring Boot `Application` class:

```
package io.reflectoring;

@SpringBootApplication
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

The `@SpringBootApplication` annotation already takes care of loading all our classes into the `ApplicationContext`.

However, we want our modules to me loosely coupled and separately testable. So we create a custom configuration class annotated with `@SpringBootConfiguration` for each module to take care of stuff that is only relevant within the respective module.

The reason why we're using `@SpringBootConfiguration` instead of the "normal" `@Configuration` annotation will become obvious later, when we're talking about testing.

The `BookingConfiguration` imports the other two configurations since it depends on them. It also enables a `@ComponentScan` for Spring beans within the module package. It also creates an instance of `BookingService` to be added to the application context:

```
package io.reflectoring.booking;

@SpringBootConfiguration
@Import({CustomerConfiguration.class, FlightConfiguration.class})
@EnableAutoConfiguration
@ComponentScan
public class BookingConfiguration {
  @Bean
  public BookingService bookingService(
          BookingRepository bookingRepository,
          CustomerRepository customerRepository,
          FlightService flightService) {
    return new BookingService(bookingRepository, customerRepository, flightService);
  }
}
```

The `CustomerConfiguration` looks similar, but it has no dependency to other configurations. Also, it doesn't provide any custom beans, since all beans are expected to be loaded via `@ComponentScan`:

```
package io.reflectoring.customer;

@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan
public class CustomerConfiguration {}
```

Let's assume that the `Flight` module contains some scheduled tasks, so we enable Spring Boot's scheduling support:

```
package io.reflectoring.flight;

@SpringBootConfiguration
@EnableAutoConfiguration
@EnableScheduling
@ComponentScan
public class FlightConfiguration {}
```

Note that each configuration class is annotated with `@EnableAutoConfiguration`. When running the application, this is actually not needed, since the `@SpringBootApplication` already brings it into play.

But when we want to start up a single module configuration in a test (without starting the whole `ApplicationContext`), we need to enable auto-configuration by hand like this.

Also note that we don't add annotations like `@EnableScheduling` at application level but instead at module level to keep responsibilities sharp and to avoid any side-effects during testing.

Now, let's have a look at what Spring Boot offers to test our layers and verticals.

# Integration Testing the Data Layer with `@DataJpaTest`

Our data layer mainly contains our JPA entities and Spring Data repositories. **Our testing efforts in this layer concentrate on testing the interaction between our repositories and the underlying database.**

Spring Boot provides the `@DataJpaTest` (https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html#boot-features-testing-spring-boot-applications-testing-autoconfigured-jpa-test) annotation to set up a stripped application context with only the beans needed for JPA, Hibernate and an embedded database:

```
@ExtendWith(SpringExtension.class)
@DataJpaTest
class CustomerRepositoryTest {

  @Autowired
  private CustomerRepository repository;

  @Test
  void findsByName() {
    Customer customer = Customer.builder()
            .name("Hurley")
            .build();
    repository.save(customer);

    List<Customer> foundCustomers = repository.findByName("Hurley");
    assertThat(foundCustomers).hasSize(1);
  }
}
```

`@DataJpaTest` goes up the package structure until it finds a class annotated with `@SpringBootConfiguration`. It then adds all Spring Data repositories within that package and all sub-packages to the application context, so that we can just autowire them and run tests against them.

**Since we have a `@SpringBootConfiguration` for each of our vertical modules, only the repositories of the one module we're currently testing will be loaded, effectively decoupling our modules even in our tests.**

A note on testing repositories: tests for repository methods only make sense for custom queries annotated with the `@Query` annotation. We don't need to test repository methods that make use of Spring Data's naming conventions (https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation) since the application won't start if we're not following the naming conventions. If we have any test that starts the application context, it will fail and notify us about our error.

Tip: if setting up the database for a complicated query becomes cumbersome, have a look at Spring Test DBUnit (https://springtestdbunit.github.io/spring-test-dbunit/) which allows to set up database content via XML files.

# Integration Testing the Web Layer with `@WebMvcTest`

Similar to `@DataJpaTest`, `@WebMvcTest` (https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html#boot-features-testing-spring-boot-applications-testing-autoconfigured-mvc-tests) sets up an application context with everything we need for testing a Spring MVC controller:

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(controllers = BookingController.class)
class BookingControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private BookingService bookingService;

  @Test
  void bookFlightReturnsHttpStatusOk() throws Exception {
    when(bookingService.bookFlight(eq(42L), eq("Oceanic 815")))
            .thenReturn(expectedBooking());

    mockMvc.perform(
            post("/booking")
                    .param("customerId", "42")
                    .param("flightNumber", "Oceanic 815"))
            .andExpect(status().isOk());
  }
}
```

Among other things, `@WebMvcTest` provides a `MockMvc` instance to the application context. We can simply inject it and use it to simulate HTTP calls against our REST controller and assert the results.

We use `@MockBean` to replace the real instance of `BookingService` with a Mockito (http://site.mockito.org/) mock object. In the test, we tell the mock object what to return to satisfy our test setup.

As before with `@DataJpaTest`, `@WebMvcTest` goes up the package structure to the first `@SpringBootConfiguration` it finds and uses it as the root for the application context.

**Thanks to our module configurations, `@WebMvcTest` will only load the application context needed for the module we're currently testing.**

# Testing ApplicationContext Startup with `@SpringBootTest`

**A must-have test for each Spring Boot application is loading the whole `ApplicationContext` once to check if the dependencies between the beans are satisfied.**

This test actually is already included in the default sources if you create your Spring Boot application via Spring Initializr (http://start.spring.io/):

```
package io.reflectoring;

@ExtendWith(SpringExtension.class)
@SpringBootTest
class ApplicationTests {

  @Test
  void applicationContextLoads() {
  }

}
```

We simply use the `@SpringBootTest` annotation which will automatically search the package structure in and above the current package for a class annotated with `@SpringBootConfiguration`.

Since every `@SpringBootApplication` is also a `@SpringBootConfiguration`, it will find our application class in the same package and start the whole application context.

If the application context cannot be started due to any configuration error or unsatisfied bean dependencies, the test will fail.

# Integration Testing a Vertical Module using `@SpringBootTest`

**If we put a test annotated with `@SpringBootTest` into the main package of one of our modules, it will create the part of the application context needed by this module** (again, because our module contains a `@SpringBootConfiguration`).

Doing this, we can now create integration tests between any beans from the application context. For example, we can create a test that goes through all layers of our module from web down to the database:

```
package io.reflectoring.booking;

@ExtendWith(SpringExtension.class)
@SpringBootTest
class BookingIntegrationTest {

  @Autowired
  private WebApplicationContext applicationContext;

  @Autowired
  private CustomerRepository customerRepository;

  private MockMvc mockMvc;

  @BeforeEach
  void setup() {
    this.mockMvc = MockMvcBuilders
            .webAppContextSetup(applicationContext)
            .build();
  }

  @Test
  void bookFlightReturnsHttpStatusOk() throws Exception {
    this.customerRepository.save(Customer.builder()
            .name("Hurley")
            .build());

    this.mockMvc.perform(
            post("/booking")
                    .param("customerId", "1")
                    .param("flightNumber", "Oceanic 815"))
            .andExpect(status().isOk());
  }

}
```

Since we're not using `@WebMvcTest` here, we have to create a `MockMvc` ourselves. And since we're not mocking the data layer anymore, we have to create a `Customer` object in the database before we call the `/booking` endpoint that reads this object from the database.

We're expecting the `customerId` to be `1` here, because we have saved exactly one object to the database and the auto-generated IDs start at `1` . Handling auto-generated IDs can quickly become ugly in test cases that need a little more database setup. Here, again, Spring Test DBUnit (https://springtestdbunit.github.io/spring-test-dbunit/) can help.

**In the test above we haven't mocked the beans coming from the other modules. If we only wanted to test the integration of the beans within our booking module, we might have used `@MockBean` to mock out the beans coming from other modules.**

# Testing single Beans with Plain Old Unit Tests

All Spring Boot test features discussed above support integration tests and not unit tests. **To implement a plain old unit test of any bean, we should actually refrain from using the Spring Boot features, because they add a significant overhead to our tests by creating an application context.**

Expecially in our business layer we want to test business logic and not integration with the web or data layer. A plain unit test might look like this:

```
package io.reflectoring.booking.business

class BookingServiceTest {

  private CustomerRepository customerRepository = Mockito.mock(CustomerRepository.class);

  private FlightService flightService = Mockito.mock(FlightService.class);

  private BookingRepository bookingRepository = Mockito.mock(BookingRepository.class);

  private BookingService bookingService;

  @BeforeEach
  void setup() {
    this.bookingService = new BookingService(
      bookingRepository,
      customerRepository,
      flightService);
  }

  @Test
  void bookFlightReturnsBooking() {
    when(customerRepository.findById(42L)).thenReturn(customer());
    when(flightService.findFlight("Oceanic 815")).thenReturn(flight());
    when(bookingRepository.save(eq(booking()))).thenReturn(booking());

    Booking booking = bookingService.bookFlight(42L, "Oceanic 815");

    assertThat(booking).isNotNull();
    verify(bookingRepository).save(eq(booking));
  }

  // factory methods customer(), flight(), and booking() omitted
}
```

Using Mockito (http://site.mockito.org/), we simply mock away all dependencies we don't want to test.

# Conclusion

This article proposed a structure for packages and configuration classes to create a modular, testable Spring Boot application. We also had a look at a few ways to test the layers and modules using Spring Boot's testing features.

If we structure the code in a sensible way, we can test our modules isolated from each other, using Spring Boot's testing features.

However, we should stick to plain unit tests if possible and only use Spring Boot's testing features as a means for integration tests. This way, we can avoid unnecessarily long-running tests that take up much time building a Spring application context.

**Tags:** boot (https://reflectoring.io/tags/#boot) junit (https://reflectoring.io/tags/#junit) spring (https://reflectoring.io/tags/#spring) testing (https://reflectoring.io/tags/#testing)

**Categories:** spring-boot (https://reflectoring.io/categories/#spring-boot)

**Updated:** May 27, 2018

---

**LEAVE A COMMENT**

**Fernando Fradegrada** • 8 months ago

This is an excellent post! You can maybe add a Spring Security Configuration and take a look on how it's involved in controller testing.

Great work!

1 • Reply • Share ›

**Mariusz Sanitariusz** • 5 months ago

I liked the idea of using @SpringBootConfiguration to denote module context scope however this says in contradiction to what the javadocs say:
* Application should only ever include *one* {@code @SpringBootConfiguration} and
* most idiomatic Spring Boot applications will inherit it from
* {@code @SpringBootApplication}.

• Reply • Share ›

**Tom Hombergs** Mod ➚ Mariusz Sanitariusz • 5 months ago

Yeah, I should warn about that in the text. I'll check with the Spring Boot Team why only one @SpringBootConfiguration should be used and add that in the text.

Thanks for the heads-up!

• Reply • Share ›