Unit Testing

Unit Testing

Dependency injection should make your code less dependent on the container than it would be with traditional J2EE / Java EE development. The POJOs that make up your application should be testable in JUnit or TestNG tests, with objects instantiated by using the `new` operator, without Spring or any other container. You can use mock objects (in conjunction with other valuable testing techniques) to test your code in isolation. If you follow the architecture recommendations for Spring, the resulting clean layering and componentization of your codebase facilitate easier unit testing. For example, you can test service layer objects by stubbing or mocking DAO or repository interfaces, without needing to access persistent data while running unit tests.

True unit tests typically run extremely quickly, as there is no runtime infrastructure to set up. Emphasizing true unit tests as part of your development methodology can boost your productivity. You may not need this section of the testing chapter to help you write effective unit tests for your IoC-based applications. For certain unit testing scenarios, however, the Spring Framework provides mock objects and testing support classes, which are described in this chapter.

Mock Objects

Spring includes a number of packages dedicated to mocking:

- Environment
- Servlet API
- Spring Web Reactive

Environment

The `org.springframework.mock.env` package contains mock implementations of the `Environment` and `PropertySource` abstractions (see Bean Definition Profiles and `PropertySource` Abstraction). `MockEnvironment` and `MockPropertySource` are useful for developing out-of-container tests for code that depends on environment-specific properties.

Servlet API

The `org.springframework.mock.web` package contains a comprehensive set of Servlet API mock objects that are useful for testing web contexts, controllers, and filters. These mock objects are targeted at usage with Spring's Web MVC framework and are generally more convenient to use than dynamic mock objects (such as EasyMock) or alternative Servlet API mock objects (such as MockObjects).

TIP

Since Spring Framework 6.0, the mock objects in `org.springframework.mock.web` are based on the Servlet 6.0 API.

MockMvc builds on the mock Servlet API objects to provide an integration testing framework for Spring MVC. See MockMvc.

Spring Web Reactive

The `org.springframework.mock.http.server.reactive` package contains mock implementations of `ServerHttpRequest` and `ServerHttpResponse` for use in WebFlux applications. The `org.springframework.mock.web.server` package contains a mock `ServerWebExchange` that depends on those mock request and response objects.

Both `MockServerHttpRequest` and `MockServerHttpResponse` extend from the same abstract base classes as server-specific implementations and share behavior with them. For example, a mock request is immutable once created, but you can use the `mutate()` method from `ServerHttpRequest` to create a modified instance.

In order for the mock response to properly implement the write contract and return a write completion handle (that is, `Mono<Void>`), it by default uses a `Flux` with `cache().then()`, which buffers the data and makes it available for assertions in tests. Applications can set a custom write function (for example, to test an infinite stream).

The WebTestClient builds on the mock request and response to provide support for testing WebFlux applications without an HTTP server. The client can also be used for end-to-end tests with a running server.

Unit Testing Support Classes

Spring includes a number of classes that can help with unit testing. They fall into two categories:

- General Testing Utilities
- Spring MVC Testing Utilities

General Testing Utilities

The `org.springframework.test.util` package contains several general purpose utilities for use in unit and integration testing.

`AopTestUtils` is a collection of AOP-related utility methods. You can use these methods to obtain a reference to the underlying target object hidden behind one or more Spring proxies. For example, if you have configured a bean as a dynamic mock by using a library such as EasyMock or Mockito, and the mock is wrapped in a Spring proxy, you may need direct access to the underlying mock to configure expectations on it and perform verifications. For Spring's core AOP utilities, see `AopUtils` and `AopProxyUtils`.

`ReflectionTestUtils` is a collection of reflection-based utility methods. You can use these methods in testing scenarios where you need to change the value of a constant, set a non-`public` field, invoke a non-`public` setter method, or invoke a non-`public` configuration or lifecycle callback method when testing application code for use cases such as the following:

- ORM frameworks (such as JPA and Hibernate) that condone `private` or `protected` field access as opposed to `public` setter methods for properties in a domain entity.
- Spring's support for annotations (such as `@Autowired`, `@Inject`, and `@Resource`), that provide dependency injection for `private` or `protected` fields, setter methods, and configuration methods.
- Use of annotations such as `@PostConstruct` and `@PreDestroy` for lifecycle callback methods.

`TestSocketUtils` is a simple utility for finding available TCP ports on `localhost` for use in integration testing scenarios.

NOTE

`TestSocketUtils` can be used in integration tests which start an external server on an available random port. However, these utilities make no guarantee about the subsequent availability of a given port and are therefore unreliable. Instead of using `TestSocketUtils` to find an available local port for a server, it is recommended that you rely on a server's ability to start on a random ephemeral port that it selects or is assigned by the operating system. To interact with that server, you should query the server for the port it is currently using.

Spring MVC Testing Utilities

The `org.springframework.test.web` package contains `ModelAndViewAssert`, which you can use in combination with JUnit, TestNG, or any other testing framework for unit tests that deal with Spring MVC `ModelAndView` objects.

TIP

Unit testing Spring MVC Controllers

To unit test your Spring MVC `Controller` classes as POJOs, use `ModelAndViewAssert` combined with `MockHttpServletRequest`, `MockHttpSession`, and so on from Spring's Servlet API mocks. For thorough integration testing of your Spring MVC and REST `Controller` classes in conjunction with your `WebApplicationContext` configuration for Spring MVC, use the MockMvc instead.