Spring Professional Exam Tutorial v5.0

Question 01

Question 01 - Do you use Spring in a unit test?

Spring Framework is usually not used in unit tests, however Spring contains some support for unit testing within following packages:

- `org.springframework.test.util`
  - `ReflectionTestUtils`
    - ORM Entities related testing – set value for private field, normally handled by ORM
    - Manual dependency injection into private field, normally handled by `@Autowired`, `@Inject`
    - `@PostConstruct` and `@PreDestroy` lifecycle callback methods testing
  - `AopTestUtils` – Aspect Oriented Programming Related Testing
- `org.springframework.test.web`
  - `ModelAndViewAssert` – Unit Testing for Spring MVC Controllers
- `org.springframework.mock.env`
  - mock implementations of the Environment and PropertySource
  - `MockEnvironment, MockPropertySource`
- `org.springframework.mock.jndi`
  - Mock implementation of JNDI SPI – usually used for Java EE
- `org.springframework.mock.web`
  - Servlet API mock objects

Question 01 - Do you use Spring in a unit test?

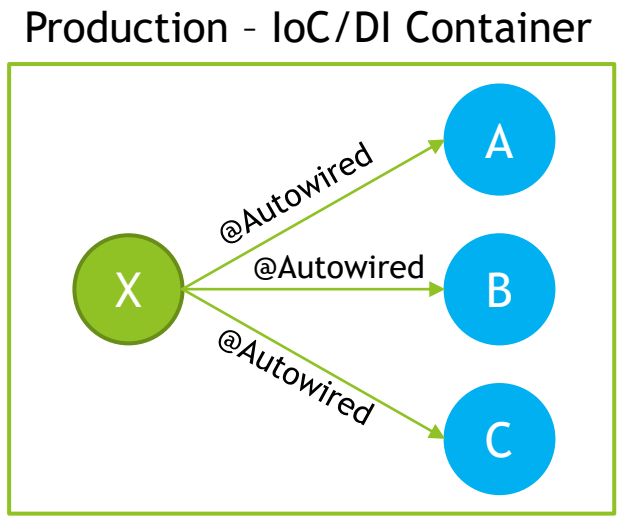
To get full understanding on why Spring is usually not used for unit tests, however some support for unit testing exists within framework, we need to get deeper into testing subject and understand differences between:

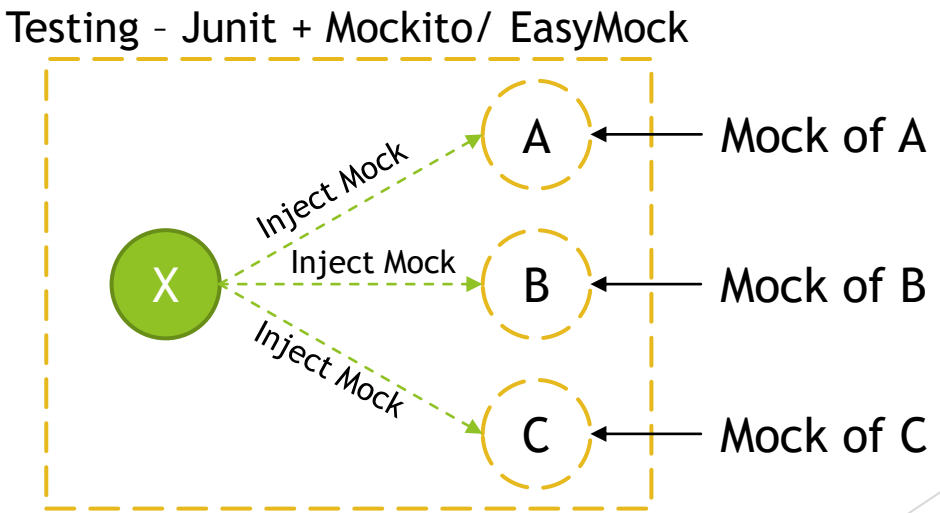
- Unit Tests
- Integration Tests
- System Tests

Having understanding on purpose and range of testing performed on each level will make it clear to know at which type of test Spring IoC/DI should be used and at which level only Spring Mocks should be used, without involving Spring IoC/DI.

Question 01 - Do you use Spring in a unit test?

Unit Tests should test one **unit of functionality** in **isolation**. This unit of functionality can be defined as **single** method, **class**, module, component. In **Object Oriented Programming**, **unit of functionality is usually defined as single class**. Testing in isolation, means that environment of execution is not initiated during testing and should not affect testing process, any dependencies on environment should be mocked. All **class collaborators should be mocked** as well. Testing should be performed **outside of container**, that means that **IoC/DI should not be required to create instances** of objects under test. Tests should execute successfully as out-of-container tests.

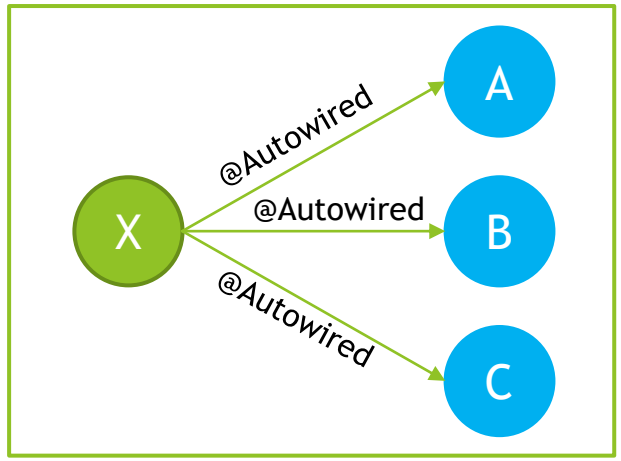




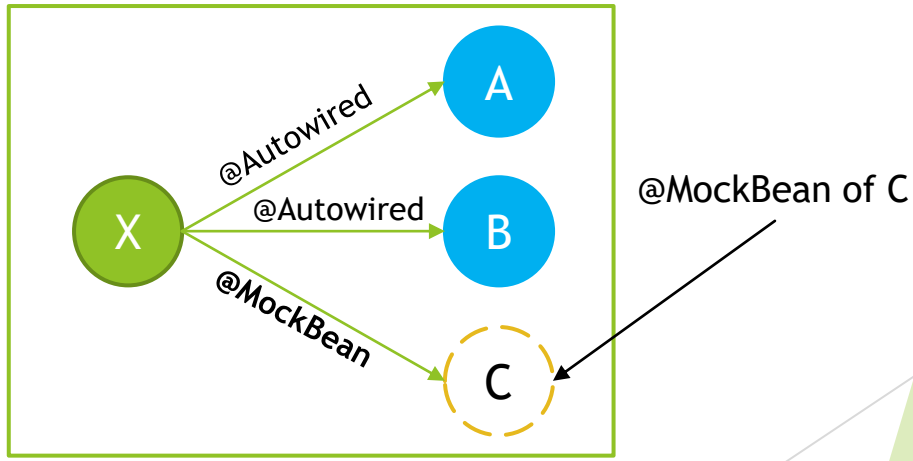
Question 01 - Do you use Spring in a unit test?

Integration Tests should test **multiple** modules or **components** that are **combined together**. Those modules should be already unit tested, and on Integration Test level some subset of functionalities should be checked, to test if modules provide requested functionalities when cooperating together under environment that should be close to production one, however with assumptions **that some of components might still be mocked**. When performing Integration Test we want to **initiate subset of system** and execute test against it. **IoC/DI Container is used** for this kind of testing, with some simplification upon deployment or container execution.

Production – IoC/DI Contianer

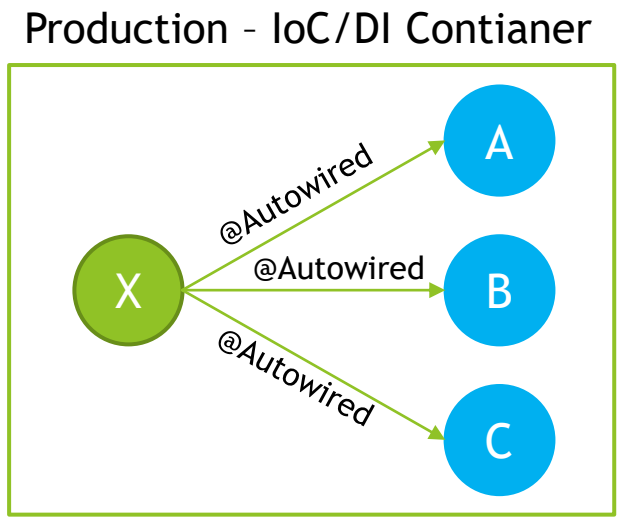


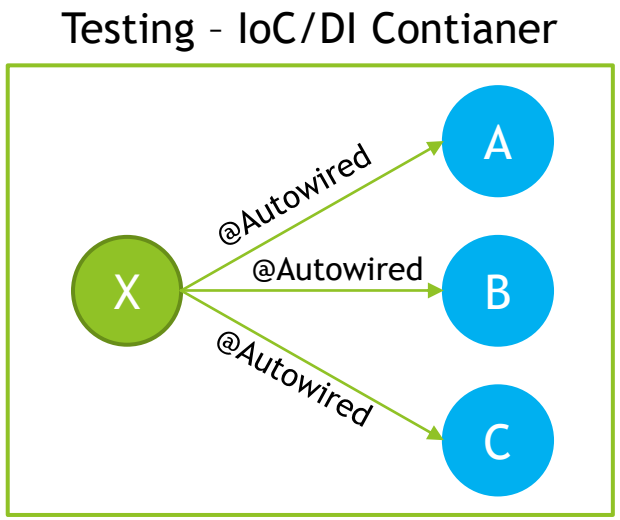
Integration Testing – @RunWith(SpringRunner.class)



Question 01 - Do you use Spring in a unit test?

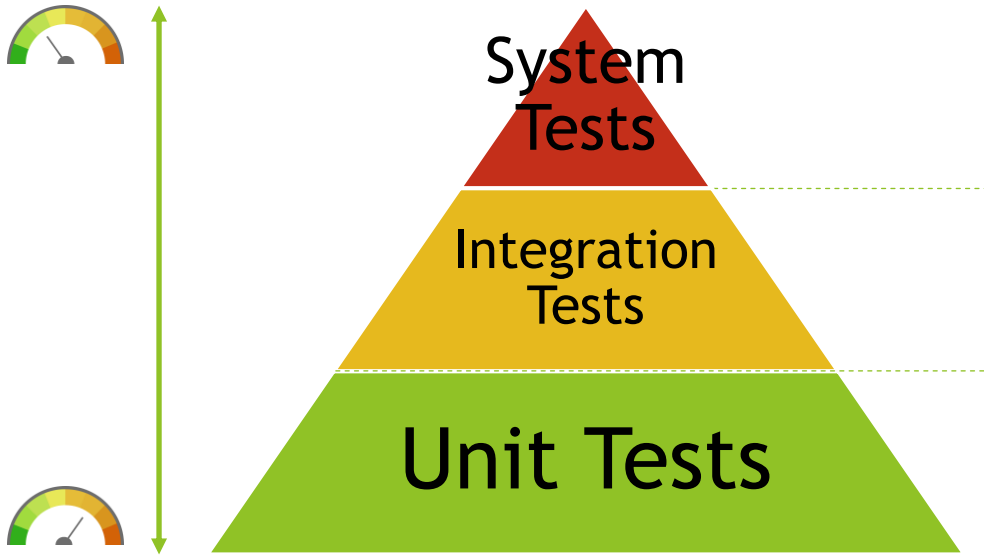
System Tests should check required system functionality on fully running system, consisting of all modules and environment that is as close as possible to production one. System under System Tests level should already be checked on unit and integration test level, and on System level only subset of functionalities should be checked to fill the gaps that were not possible to test on unit and integration testing level. **IoC/DI container is used** with assumption that it is used in **the same way as production environment**.





Question 01 - Do you use Spring in a unit test?

In practice, large scale enterprise systems needs to have testing strategy defined to test system that is being developed in efficient manner. Often used strategy for test automation is "Testing Pyramid". It assumes that Unit, Integration and System Tests are implemented, with right ratio on number of tests on each level.



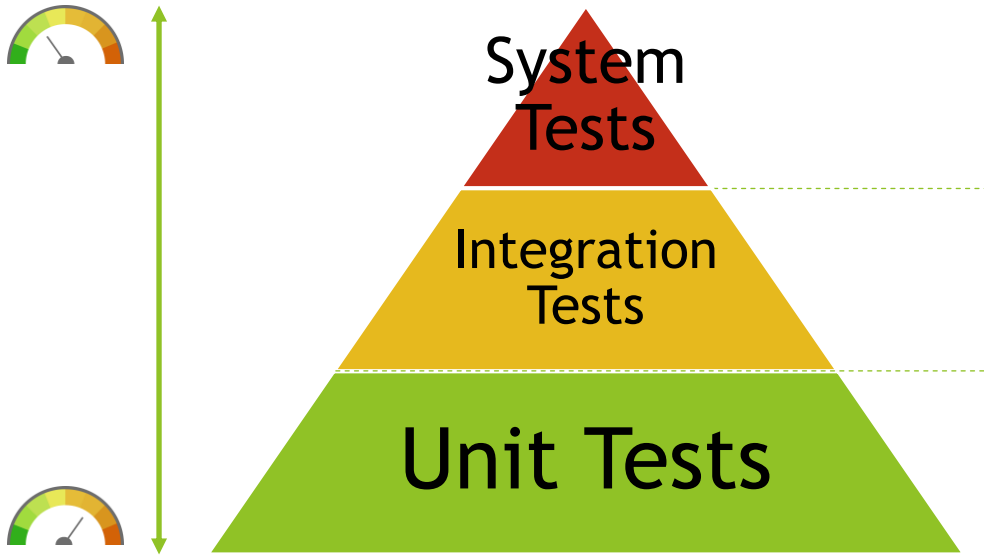
$$$
- **Execution:** Slow
- **Maintenance:** Expensive
- Easy to test System Cases
- Hard to test Detailed Cases

- **Execution:** Medium-to-Slow
- **Maintenance:** Medium-to-High cost
- Details testing is a challenge
- Easy to test Integration between components

$
- **Execution:** Fast
- **Maintenance:** Cheap
- Hard or impossible to test Integration/System Cases
- Easy to test Detailed Cases

Question 01 - Do you use Spring in a unit test?

On each level right set of tools should be used to perform testing, minimize costs and maximize feedback from automated tests.



$$$
- Automated Provisioning
- Automated Deployment
- Automated Test Execution Against Environment
- Spring Boot Executable Jar / Ansible / Containers

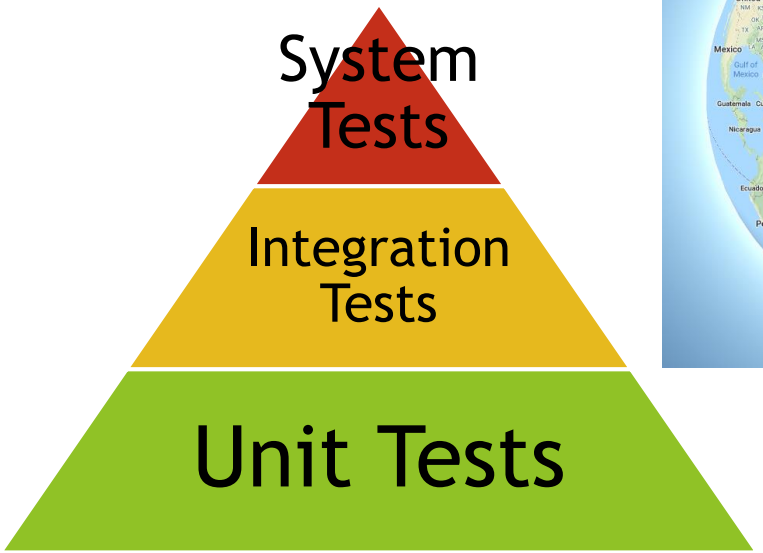
- **IoC/DI for Integration Testing**
- **@RunWith(SpringRunner.class)**
- @MockBean - Mockito

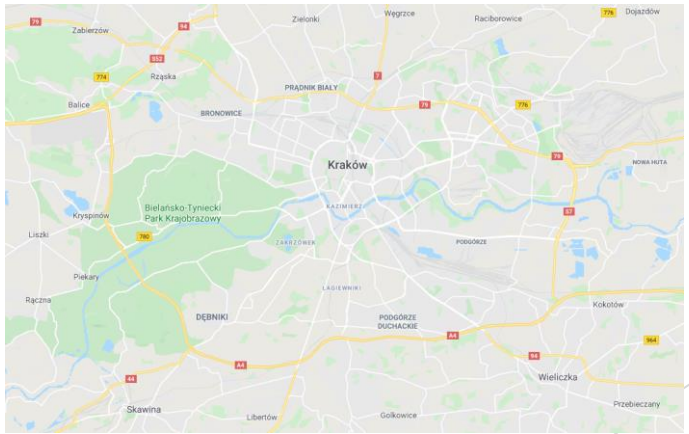
- **Junit**
- **Mockito/ EasyMock**

$

Question 01 - Do you use Spring in a unit test?

In large scale projects, you need to have all types of tests implemented, as on each level you can look at different level of details of the system.





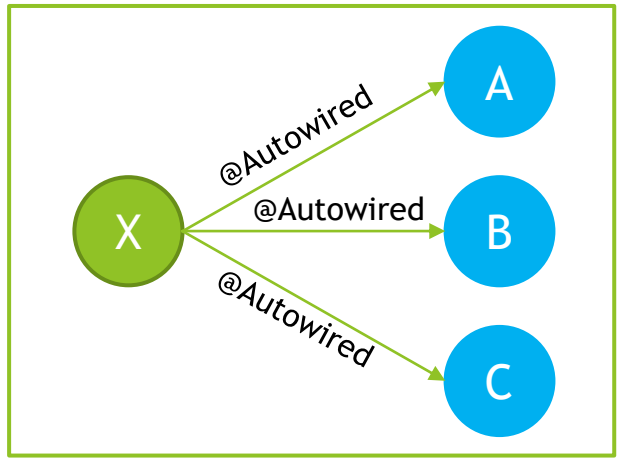
Spring Professional Exam Tutorial v5.0

Question 02

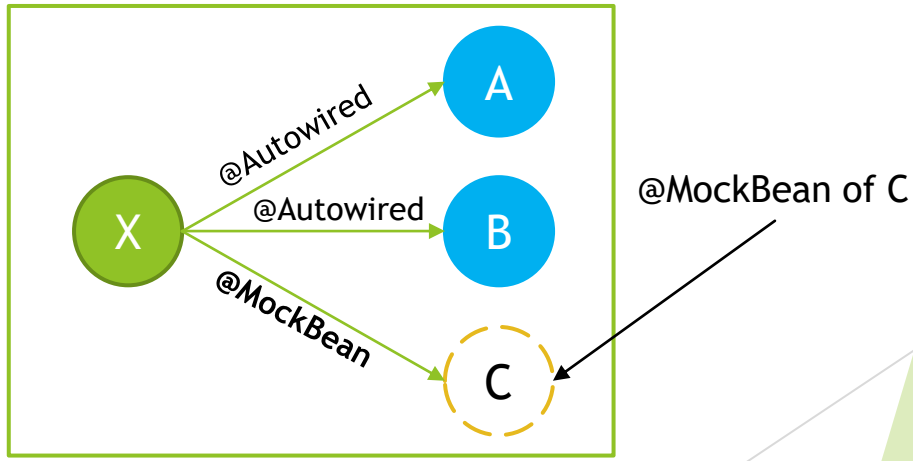
Question 02 - What type of tests typically use Spring?

Integration Tests are type of tests that typically use Spring. Reason for it is that on Integrate Test level we want to test multiple **components** that are **combined together**, and we want to check if those components provide requested functionalities when cooperating together under environment that should be close to production one, however with assumptions **that some of components might still be mocked.** When performing Integration Test we want to **initiate subset of system** and execute test against it. **IoC/DI Container is used** for this kind of testing, with some simplification upon deployment or container execution. **Dependencies are resolved and injected by Spring**.

Production – IoC/DI Container



Integration Testing – @RunWith(SpringRunner.class)



Question 02 - What type of tests typically use Spring?

Spring provides great support for integration testing, with main goals of the support being:

- Management of Spring IoC container and IoC container caching between tests
  - Spring will create and manage IoC container for tests
  - Context can be reused between tests
  - Main purpose of context being reused is to improve tests execution time
  - Execution times of integration tests might be long mainly because of Embedded Database, Hibernate and other components that are created once context is created

- Dependency Injection in tests
  - Allows for easy Spring Test definition with usage of `@RunWith(SpringRunner.class)`
  - `@ContextConfiguration` can be used in tests to configure context
  - Support for `@Autowired`, `@Inject` ...
  - Allows customization with `@TestExecutionListener`

- Transaction management appropriate to integration testing
  - Resolves issue with test affecting each other on data level by implementing proper transaction management
  - By default, all transactions are roll-back transactions
  - Gives ability to commit transactions if required

Question 02 - What type of tests typically use Spring?

Additionally Spring provides following tools to simplify Integration Testing:

- JDBC Testing Support
  - `JdbcTestUtils` provides JDBC related utility functions
  - `countRowsInTable`, `countRowsInTableWhere`, `deleteFromTables`, `deleteFromTableWhere`, ...
- Spring MVC Testing Support
  - Allow for easy setup with:
    - `@RunWith(SpringRunner.class)`
    - `@WebAppConfiguration`
    - `@ContextConfiguration`
  - Automatically creates `MockMvc`
- HtmlUnit Integration
  - Simplifies end-to-end testing for HTML views
- Client-Side REST Tests
  - Allows you to test Client code that interacts with mocked REST Service
  - Use `RestTemplate` with `MockRestServiceServer` to make assertions on mock

Question 02 - What type of tests typically use Spring?

Spring annotations for Integration Testing

- @ContextConfiguration
  - Allows you to specify how to load and configure an ApplicationContext for integration tests
  - You can specify `@Configuration` classes that will be used during `ApplicationContext` loading
  - Optionally, you can specify XML configuration files locations, if you are using it instead of annotated `@Configuration` classes

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = ApplicationConfiguration.class)
public class ApplicationServiceIntegrationTest {

    ...

}
```

- @BootstrapWith
  - Allows for low-level control on how Context for Tests is created
  - To implement custom bootstrapped, it is best to extend `AbstractTestContextBootstrapper`
  - Used at class-level

Question 02 - What type of tests typically use Spring?

Spring annotations for Integration Testing

- @DirtiesContext
  - Marks test as one that modifies state of context, and it means that context should be recreated prior next test execution because otherwise modified context state might affect test execution

```
@Test
@DirtiesContext
public void shouldBookAnyRoomForNewGuest() {
    ...
}
```

  - When used at class-level you can specify following modes:
    - BEFORE_CLASS
    - BEFORE_EACH_TEST_METHOD
    - AFTER_EACH_TEST_METHOD
    - AFTER_CLASS

```
@DirtiesContext(classMode = AFTER_EACH_TEST_METHOD)
```

  - When used at method-level you can specify following modes:
    - BEFORE_METHOD
    - AFTER_METHOD

```
@DirtiesContext(methodMode = AFTER_METHOD)
```

Question 02 - What type of tests typically use Spring?

Spring annotations for Integration Testing

- @ActiveProfiles
  - class-level annotation that is used to declare which bean definition profiles should be active when loading an ApplicationContext

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = ApplicationConfiguration.class)
@ActiveProfiles({"test", "example-data"})
public class ApplicationServiceIntegrationTest {

    ...

}
```

- @TestPropertySource
  - class-level annotation that you can use to configure the locations of properties files and inlined properties

```
@TestPropertySource("/application-test.properties")
```

```
@TestPropertySource(properties = { "user = test-user", "group = test-group" })
```

Question 02 - What type of tests typically use Spring?

Spring annotations for Integration Testing

- @WebAppConfiguration
  - class-level annotation that triggers creation of `MockServletContext`, which serves as the `ServletContext` for the test's `WebApplicationContext`
  - Indicates that `ApplicationContext` loaded for an integration test should be a `WebApplicationContext`

```
@RunWith(SpringRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class ApplicationServiceIntegrationTest {

    ...

}
```

- @ContextHierarchy
  - Used when hierarchy of application contexts has to be used for integration test

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextHierarchy({
        @ContextConfiguration(classes = AppConfig.class),
        @ContextConfiguration(classes = WebConfig.class)
})
public class WebIntegrationTests {
    ...
}
```

Question 02 - What type of tests typically use Spring?

Spring annotations for Integration Testing

- @TestExecutionListeners
  - Allows registration of `TestExecutionListener` which allows for customization of test execution
  - Example of `TestExecutionListener` that is registered by default is `DirtiesContextTestExecutionListener`

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = ApplicationConfiguration.class)
@TestExecutionListeners({
        CustomTestExecutionListener1.class,
        CustomTestExecutionListener2.class
})
public class ApplicationServiceIntegrationTest {

    ...

}
```

- @Commit
  - class or method level annotation
  - indicates that after test execution, transaction should be committed

```
@Test
@Commit
public void shouldBookAnyRoomForNewGuest() {
    ...
}
```

Question 02 - What type of tests typically use Spring?

Spring annotations for Integration Testing

- @Rollback
  - class or method level annotation that indicates that transaction should be rolled back after test execution
  - Even if `@Rollback` is not explicitly defined, all transactions under tests will be rolled backed by default

```
@Test
@Rollback
public void shouldBookAnyRoomForNewGuest() {
    ...
}
```

- @BeforeTransaction
  - Indicates method that should be executed before transaction is started
- @AfterTransaction
  - Indicates method that should be executed after transaction is started

Question 02 - What type of tests typically use Spring?

Spring annotations for Integration Testing

- @Sql
  - Indicates SQL scripts that should be executed against database during integration test

```
@Test
@Sql({
        "/test-schema.sql",
        "/test-data.sql"
})
public void shouldBookAnyRoomForNewGuest() {
    ...
}
```

- @SqlConfig
  - Defines metadata used for SQL script parsing

```
@Test
@Sql(
        scripts = "/test-user-data.sql",
        config = @SqlConfig(commentPrefix = "`", separator = "@@")
)
public void shouldBookAnyRoomForNewGuest() {
    ...
}
```

Question 02 - What type of tests typically use Spring?

Spring annotations for Integration Testing

- @SqlGroup
  - Allows you to use multiple @Sql annotations

```
@Test
@SqlGroup({
        @Sql(scripts = "/test-schema.sql", config = @SqlConfig(separator = "@@")),
        @Sql("/test-data.sql")
})
public void shouldBookAnyRoomForNewGuest() {
    ...
}
```

Spring Professional Exam Tutorial v5.0

Question 03

Question 03 - How can you create a shared application context in a JUnit integration test?

"Shared Application Context" can be considered as:

- Sharing Context **Definition**
- Sharing Context **Instance**

Context **Definition** can be shared between tests in following way:

- Use **Base Class** for All Tests which will contain `@ContextConfiguration` and other annotations, like for example `@ActiveProfiles` etc.
- Use **Custom Annotation** that will contain context configuration
- Use **Test Configuration that Inherits Application Configuration**
- Use **Base Interface** for all Tests that will contain context configuration

Context **Instance** is **shared by default between all tests**, as long as requested context matches one that is already cached.

Context will be reused as long as `@ContextConfiguration` (locations, classes, initializers, ...), `@ActiveProfiles`, `@TestPropertySource` attributes are matched. To see full list of attributes that needs to be matched, you can look at `MergedContextConfiguration` class that is used as key to `ContextCache`.

You can use `@DirtiesContext` annotation to force Spring to create new instance of context for test.

Spring Professional Exam Tutorial v5.0

Question 04

Question 04 - When and where do you use @Transactional in testing?

When - you use `@Transactional` annotation in testing whenever you want to run some part of the code that can alter state of transactional resource, for example database. Usage of this annotation allows you to mark code that should execute under transaction and allows to rollback all changes made by test, allowing other tests to pick from clear state.

By default, transaction will be rolled-back for each test which was executed with `@Transactional` annotation.

You can gain additional control over transaction when using `@Transactional` annotation with:

- `@Rollback` - explicitly state to rollback transaction
- `@Commit` - indicates that after test execution, transaction should be committed
- `@BeforeTransaction` - indicates method that should be executed before transaction is started
- `@AfterTransaction` - indicates method that should be executed after transaction is started

Question 04 - When and where do you use @Transactional in testing?

Where - `@Transactional` can be used:

- On top of the class – each test method in class will be executed in transaction

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = ApplicationConfiguration.class)
@Transactional
public class ApplicationServiceIntegrationTest {

    ...

}
```

- On top of the method – test method will be executed in transaction

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = ApplicationConfiguration.class)
public class ApplicationServiceIntegrationTest {

    @Transactional
    @Test
    public void shouldRegisterGuests() {
        ...
    }
}
```

Spring Professional Exam Tutorial v5.0

Question 05

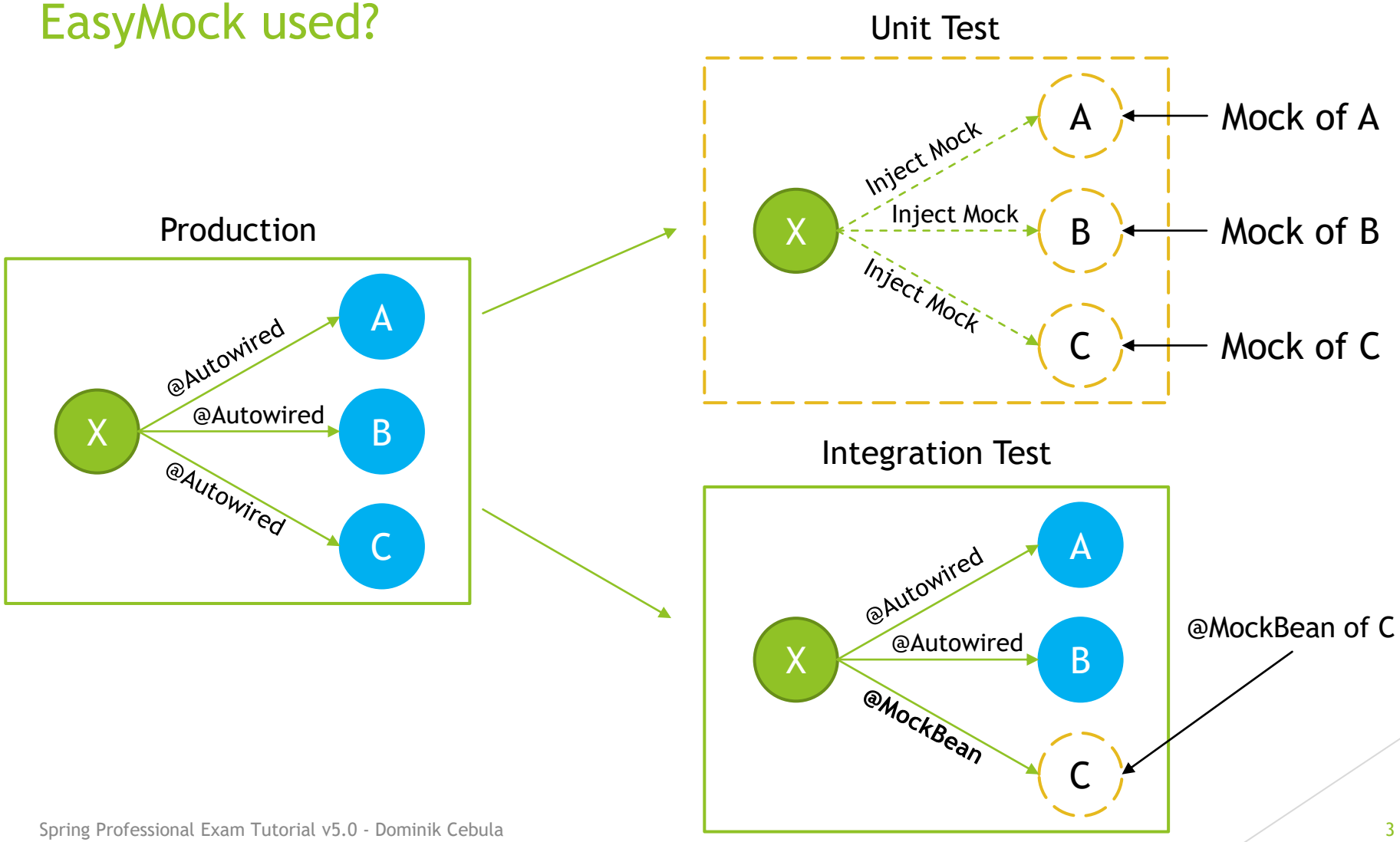
Question 05 - How are mock frameworks such as Mockito or EasyMock used?

Mock Frameworks like Mockito or EasyMock are used mainly during Unit Testing to mock collaborators of classes under test. Mockito or EasyMock can be also used during Integration Testing when goal is to check cooperation between different objects, while still mocking part of the system.

Mock created with Mockito or EasyMock is a dynamic object, which can "pretend" real object and return predefined results when invoking method on it. Additionally Mock allows you to verify if expected method were indeed called with expected arguments.

Above frameworks also allows you to inject mocks to classes under test in convenient way, with usage of annotations, with style similar to IoC/DI without having to run within container at all, which is one of the reason why unit tests are so fast and lightweight.

Question 05 - How are mock frameworks such as Mockito or EasyMock used?



Question 05 - How are mock frameworks such as Mockito or EasyMock used?

Mockito usage in Unit Test – for full documentation go to https://site.mockito.org/

```
@RunWith(MockitoJUnitRunner.class)
public class ApplicationServiceTest {

    ...

    @InjectMocks
    private ApplicationService applicationService;
    @Mock
    private GuestRegistrationService guestRegistrationService;
    @Mock
    private BookingService bookingService;

    ...

    @Test
    public void shouldBookRoomAfterRegisteringUserAndConfirmingRoomAvailability() {
        when(bookingService.findAvailableRoom(DATE_2020_JULY_20)).thenReturn(Optional.of(room));
        when(guestRegistrationService.registerGuest(GUEST_TO_REGISTER)).thenReturn(registeredGuest);
        when(bookingService.bookRoom(room, registeredGuest, DATE_2020_JULY_20)).thenReturn(Optional.of(reservation));

        BookingResult bookingResult = applicationService.bookAnyRoomForNewGuest(JOHN, DOE, DATE_2020_JULY_20);

        verify(guestRegistrationService).registerGuest(GUEST_TO_REGISTER);
        verify(bookingService).bookRoom(room, registeredGuest, DATE_2020_JULY_20);

        assertEquals(ROOM_BOOKED, bookingResult.getBookingState());
        assertEquals(Optional.of(reservation), bookingResult.getReservation());
    }
}
```

Specify Mockito Runner that will handle annotations

Specify Object under test to which mocks will be injected

Mocks

Stub Answers with when(...)

Execute code

verify(...) interactions

Question 05 - How are mock frameworks such as Mockito or EasyMock used?

Mockito usage in Integration Test – prefer usage of `@MockBean` is using Spring Boot

```java
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = ApplicationConfiguration.class)
public class ApplicationServiceIntegrationTest {

    @Autowired
    private ApplicationService applicationService;
    @Autowired
    private RoomRepository roomRepository;
    @Autowired
    private ReservationRepository reservationRepository;

    @Autowired
    private GuestSharableDataService guestSharableDataServiceMock;

    @Test
    public void shouldFetchGuestSharableData() {
        when(guestSharableDataServiceMock.getGuestSharableData()).thenReturn(SHARABLE_DATA);

        String guestSharableData = applicationService.getGuestSharableData();

        verify(guestSharableDataServiceMock).getGuestSharableData();

        assertEquals(SHARABLE_DATA, guestSharableData);
    }

    @Configuration
    public static class ApplicationServiceIntegrationTestMockConfiguration {

        @Bean
        public GuestSharableDataService guestSharableDataService() {
            return mock(GuestSharableDataService.class);
        }
    }
}
```

Use Spring IoC/DI for Integration Test

Inject all dependencies from context

Inject Mock from Configuration with Mock

Create Mock manually as `@Bean`

Question 05 - How are mock frameworks such as Mockito or EasyMock used?

EasyMock usage in Unit Test – for full documentation go to https://easymock.org/

```
@RunWith(EasyMockRunner.class)
public class ApplicationServiceTest {

    ...

    @TestSubject
    private ApplicationService applicationService;
    @Mock
    private GuestRegistrationService guestRegistrationService;
    @Mock
    private BookingService bookingService;

    ...

    @Test
    public void shouldBookRoomAfterRegisteringUserAndConfirmingRoomAvailability() {
        expect(bookingService.findAvailableRoom(DATE_2020_JULY_20)).andReturn(Optional.of(room));
        expect(guestRegistrationService.registerGuest(GUEST_TO_REGISTER)).andReturn(registeredGuest);
        expect(bookingService.bookRoom(room, registeredGuest, DATE_2020_JULY_20)).andReturn(Optional.of(reservation));

        replay(guestRegistrationService, bookingService);

        BookingResult bookingResult = applicationService.bookAnyRoomForNewGuest(JOHN, DOE, DATE_2020_JULY_20);

        verify(guestRegistrationService, bookingService);

        assertEquals(ROOM_BOOKED, bookingResult.getBookingState());
        assertEquals(Optional.of(reservation), bookingResult.getReservation());
    }
}
```

Specify EasyMock Runner that will handle annotations

Specify Object under test to which mocks will be innjected

Mocks

Stub Answers with expect(…)

Execute code

verify(...) interactions

Question 05 - How are mock frameworks such as Mockito or EasyMock used?

EasyMock usage in Integration Test

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = ApplicationConfiguration.class)
public class ApplicationServiceIntegrationTest {

    @Autowired
    private ApplicationService applicationService;
    @Autowired
    private RoomRepository roomRepository;
    @Autowired
    private ReservationRepository reservationRepository;

    @Autowired
    private GuestSharableDataService guestSharableDataServiceMock;

    @Test
    @DirtiesContext
    public void shouldFetchGuestSharableData() {
        expect(guestSharableDataServiceMock.getGuestSharableData()).andReturn(SHARABLE_DATA);

        replay(guestSharableDataServiceMock);

        String guestSharableData = applicationService.getGuestSharableData();

        verify(guestSharableDataServiceMock);

        assertEquals(SHARABLE_DATA, guestSharableData);
    }

    @Configuration
    public static class ApplicationServiceIntegrationTestMockConfiguration {

        @Bean
        public GuestSharableDataService guestSharableDataService() {
            return mock(GuestSharableDataService.class);
        }
    }
}
```

Use Spring IoC/DI for Integration Test

Inject all dependencies from context

Inject Mock from Configuration with Mock

Create Mock manually as `@Bean`

Spring Professional Exam Tutorial v5.0

Question 06

Question 06 - How is @ContextConfiguration used?

`@ContextConfiguration` annotation is used on top of the class that represents Integration Test, and it's purpose is to specify how to load and configure Application Context for Integration Test.

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = ApplicationConfiguration.class)
public class ApplicationServiceIntegrationTest {

    ...

}
```

`@ContextConfiguration` annotation can be used in two basic modes:

- Annotated Classes Based Approach
- XML Based Approach

Question 06 - How is @ContextConfiguration used?

When `@ContextConfiguration` annotation is used with Annotated Classes Approach, it is expected for `classes` field to point to class or list of classes that represents `@Configuration` classes, which will contain information about `@ComponentScan` or will provide `@Bean`(s).

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = ApplicationConfiguration.class)
public class ApplicationServiceIntegrationTest {

    ...

}
```

```
@Configuration
@ComponentScan("com.app.example")
public class ApplicationConfiguration {

    ...

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
                .generateUniqueName(true)
                .build();
    }

    ...

}
```

Question 06 - How is @ContextConfiguration used?

When `@ContextConfiguration` annotation is used with XML Based Approach, it is expected for `locations` field to point to context xml definitions or list of context xml definitions.

```
@RunWith(SpringRunner.class)
@ContextConfiguration(locations = "/application-configuration.xml")
public class ApplicationServiceIntegrationTest {

    ...

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.app.example"/>

</beans>
```

Question 06 - How is @ContextConfiguration used?

Additionally `@ContextConfiguration` annotation also allows you to specify:

- initializers - list of `ApplicationContextInitializer`, used within cases that require some programmatic initialization of the application context
- loader - usually not used and default `DelegatingSmartContextLoader` is used, if required, this field allows you to specify custom context loader or one of the following:
  - GenericXmlContextLoader
  - GenericGroovyXmlContextLoader
  - AnnotationConfigContextLoader
  - GenericXmlWebContextLoader
  - GenericGroovyXmlWebContextLoader
  - AnnotationConfigWebContextLoader
  - ...
- name - name of the context hierarchy level represented by this configuration, only applicable when used within a test class hierarchy configured using `@ContextHierarchy`
- inheritLocations - whether or not xml locations or annotated classes from test superclasses should be inherited, default true
- inheritInitializers - whether or not context initializers from test superclasses should be inherited, default true

Spring Professional Exam Tutorial v5.0

Question 07

Question 07 - How does Spring Boot simplify writing tests?

Spring Boot simplifies writing tests in following way:

- Provides `@SpringBootTest` annotation - alternative to `@ContextConfiguration`, creates `ApplicationContext` through `SpringApplication`, Enables Tests Auto-Configuration, Enables Spring Boot Test Features
- Provides `@MockBean` annotation – easy creation and injection of Mockito mock

@SpyBean allows selection of method to stub vs. @MockBean that stubs all method

- Provides `@SpyBean` annotation - easy creation and injection of Mockito spy
- Provides `@WebMvcTest` annotation - useful when test focuses only on Spring MVC components, disables full auto-configuration and applies only configuration relevant to MVC tests

, test controllers while all the other components behind the controllers are mocked. RestAssured is a framework that allows even deeper testing of the presentation layer

- Provides Web Environments
  - MOCK (default)
  - RANDOM_PORT
  - DEFINED_PORT
  - NONE

Question 07 - How does Spring Boot simplify writing tests?

- Provides algorithm for Tests Environment Auto-Configuration
  - Based on defined dependencies, beans, properties, resources provides beans necessary for integration tests
  - Allows you to focus on test content instead of focusing on how to configure specified technology for integration test

- Allows to explicitly use Auto-Configurations:
  - `@JsonTest` - Auto-configured JSON Tests
  - `@WebMvcTest` - Auto-configured Spring MVC Tests (context limited to MVC)
  - `@JdbcTest` - Auto-configured JDBC Tests
  - `@DataJpaTest` - Auto-configured Data JPA Tests
  - `@JooqTest` - Auto-configured jOOQ Tests
  - `@DataMongoTest` - Auto-configured Data MongoDB Tests
  - `@RestClientTest` - Auto-configured REST Clients
  - ...

Question 07 - How does Spring Boot simplify writing tests?

- Provides `spring-boot-starter-test` module, which includes:
  - JUnit
  - Spring Test
  - Spring Boot Test
  - AssertJ - fluent assertion library.
  - Hamcrest - library of matcher objects
  - Mockito - mocking framework.
  - JSONassert - An assertion library for JSON
  - JsonPath - XPath for JSON

- Provides `@Conditional` annotations
  - `@ConditionalOnClass`
  - `@ConditionalOnMissingBean`
  - `@ConditionalOnProperty`
  - `@ConditionalOnResource`
  - `@ConditionalOnExpression`

Spring Professional Exam Tutorial v5.0

Question 08

Question 08 - What does @SpringBootTest do? How does it interact with @SpringBootApplication and @SpringBootConfiguration?

`@SpringBootTest` annotation provides following features over regular Spring Test Context:

- Automatically searches for `@SpringBootConfiguration`
  - ...unless nested `@Configuration` is detected or explicit `@SpringBootTest(classes=...)` is specified
  - In most cases `@SpringBootConfiguration` is not explicitly used, it is inherited from `@SpringBootApplication` used in production code to indicate starting place for application
  - `@SpringBootConfiguration` is an alternative to standard `@Configuration`, advantage is that `@SpringBootConfiguration` can be found automatically in tests
- Sets default `ContextLoader` to `SpringBootContextLoader`
  - ...unless one is explicitly specified in `@ContextConfiguration(loader=...)`
  - `SpringBootContextLoader` is specific `ContextLoader` that starts tests using `SpringApplication`
- Provides Web Environments
  - MOCK (default)
  - RANDOM_PORT
  - DEFINED_PORT
  - NONE
- Allows to easily define `Environment` properties
  - `properties` field of `@SpringBootTest` annotation can be used to define `key=value` pairs that will be added to `Environment` before tests execution
- Registers `TestRestTemplate` and `WebTestClient`

Question 08 - What does @SpringBootTest do? How does it interact with @SpringBootApplication and @SpringBootConfiguration?

`@SpringBootTest` annotation allows you to set following fields:

- `classes` - annotated classes to use for loading an `ApplicationContext`
  - If not set, `@SpringBootTest` will automatically search for `@SpringBootConfiguration` which is usually inherited from `@SpringBootApplication`
- `properties - key=value` pairs that will be added to `Environment` before tests execution
- `webEnvironment` – one of specified web environment used for web layer testing:
  - MOCK (default)
  - RANDOM_PORT
  - DEFINED_PORT
  - NONE

Question 08 - What does @SpringBootTest do? How does it interact with @SpringBootApplication and @SpringBootConfiguration?

`@SpringBootTest` annotation interacts with `@SpringBootApplication` and `@SpringBootConfiguration` through `SpringBootTestContextBootstrapper` and `SpringBootContextLoader`.

Goal of `SpringBootContextLoader` is to transform initial `ContextConfiguration` to `ApplicationContext`. `SpringBootContextLoader` will get as input `class` annotated with `@SpringBootConfiguration`, which will be located by `SpringBootTestContextBootstrapper`.

Algorithm that searches for class annotated with `@SpringBootConfiguration` will start at package where test class is located and will scan this package and all parent packages in search for `@SpringBootApplication`.

Question 08 - What does @SpringBootTest do? How does it interact with @SpringBootApplication and @SpringBootConfiguration?

`@SpringBootTest` annotation is used in following way:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ApplicationServiceIntegrationTest {

    ...

}
```

Discovers

```
@SpringBootApplication
public class Runner {

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

```
...
@SpringBootConfiguration
...
public @interface SpringBootApplication {
...
```

`@SpringBootApplication` definition

`@SpringBootApplication` inherits `@SpringBootConfiguration`