EInnovator
Software Engineering School

13

Spring Testing

Jorge Simão, Ph.D.

Content

Spring Test Overview

Spring Framework provides a well-rounded number of mechanisms to support application testing according to the principles of test-driven development. **Spring Test** framework automatically setups the Spring **ApplicationContext** to be used during tests, and allows dependency-injection to be applied to test classes. Integration with **JUnit4** and **TestNG** testing frameworks is supported, as well as standalone and web application testing. Annotation-driven meta-data can be use to tailor the test configuration and actions performed during the execution of tests using POJO test classes. Specific mechanisms are available to test transactional database code and avoid interference of test methods. For web-development and Spring MVC applications, out-of-the-container testing is supported to increase developer productivity and simplify the automation of integration testing pipelines.

To get started with **Spring Testing** module the **Maven**/**Gradle** dependency **spring-test** should be imported. Alternatively, for **Spring Boot** applications, use dependency **spring-boot-starter-test**, which also imports **JUnit4** and **Mockito** mocking framework.

» Example: Spring Test Maven Dependency Import

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>4.3.3.RELEASE</version>
</dependency>
```

Example: Importing Spring Boot Test Starter

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <version>1.4.1.RELEASE</version>
</dependency>
```

Test-Driven Development

Test-Driven development is the practice of creating dedicated test classes as separated project artefacts and running them to test the validity of business code.

This is considered a recommended/mandatory best-practice in order to build reliable software. The test classes allow for the automation and repeatability of the testing procedure, which is essential to streamline development – specially when complexity grows and code is continuously being refactored, as is the case with agile software development methodologies.

Types of Tests

The different types of software tests performed in enterprise systems are frequently classified in the several categories. **Spring Test** framework give support most strongly for to integration and out-of-the-container tests.

- **Unit Tests** – Testing of a single unit of functionality such as a Java class or method. Dependency are expected to be kept as minimal as possibles.
- **Integration Tests** – Testing of multiple components working together as an integrated sub-system. Simplifies the testing of components and sub-systems as non-functional requirements might be ignore (e.g. performance optimization and security).
- **Out-of-the-Container Tests** – Web application and REST-WS are tested in with full configuration without requirement deployment on a container.
- **End-to-End Tests** – Client-Server interaction Tests.
- **User Acceptance Tests** – User-centric tests, such as UI features and overall user-experience.
- **Performance, Scalability, Reliability Tests** – Testing the behaviour of the system under conditions of stress, such as high-load and failing components.
- **Security Tests** – Checks for the possibility of system intrusions and user exploits.

JUnit4 & TestNG Integration

Spring Test integrates with **JUnit4** in a seemly way, by using **JUnit4** annotation **@RunWith** to specify a **Runner** class (driver) for the tests. This is the class that is responsible to create and initialize instances of the test class and call the test methods. Spring provide the class **SpringJUnit4ClassRunner** for this purpose, or since version 4.3 the shorter alias class **SpringRunner**. The use of the **@RunWith** annotation allow for the test

Spring Testing

classes to defined as POJOs. Alternatively, JUnit4 tests classes can extend the provided abstract class **AbstractJUnit4SpringContextTests**, or when running transactional methods the class **AbstractTransactionalJUnit4SpringContextTests.** With either approach the same set of features of the **Spring Test** framework is or can be enabled.

Spring Test also integrates with **TestNG** testing framework. In this case, test class should extend the provided abstract class **AbstractTestNGSpringContextTests**. These abstract classes, both for JUnit4 and TestNG, define and initialize several protected fields – such as the underlying **ApplicationContext** – that can be used by test classes programmatically (e.g. in a **@Before** setup method in the case of JUnit4).

» Example: Running JUnit4 Tests with a Spring Runner

```
@RunWith(SpringJUnit4ClassRunner.class)
public class OrderServiceTests {

  @Test
  public void cancelOrderTest() { ... }
}
```

Test ApplicationContext

The **Spring Test** framework automatically starts a Spring **ApplicationContext** that creates, initializes, and manages application components (beans) for the execution of the tests. The annotation **@ContextConfiguration** defines which configuration resources define the application components (beans) to use for tests. The **value()** or **locations()** attribute specifies XML bean files to load, while the **classes()** attribute specifies **@Configuration** classes or component classes with stereotype annotations (e.g. **@Component**, **@Service**, **@Repository**, **@Controller**, and **@RestController**). Static inner **@Configuration** classes are also searched inside the test class. If no configuration resource is specified in either of these ways, a default XML file is assumed with name ***TestClass*-context.xml**.

The different types of configuration resources are exclusive. To combine them, a single type should be used to bootstrap the **ApplicationContext** and make an import a posteriori (e.g. if bootstrapping with XML files, the annotation **@ComponentScan** can be used; if bootstrapping with **@Configuration** classes use **@ImportResource** to load XML files).

» Example: Testing with XML Configuration Files

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations=
  {"app-config.xml", "test-config.xml"})
public class OrderServiceTests {
  ...
}
```

» Example: Testing with Java Config Classes

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=
  {AppConfig.class, TestConfig.class})
public class OrderServiceTests {
  ...
}
```

» Example: Testing with Inner Java Config Classes

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class OrderServiceTests {
  @Configuration
  @Import(AppConfig.class)
  static class TestConfig {
    @Bean DataSource dataSource() {
      return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("classpath:schema.sql")
        .addScript("classpath:test-data.sql")
        .ignoreFailedDrops(true)
        .build();
    }
  }

  @Test
  public void cancelOrderTest() { ... }
}
```

Spring Test further facilitates testing by allowing dependency-injection to be applied to instance of the test classes – e.g. via the **@Autowired**, **@Inject**, **@Resource** annotation. This is useful to inject configured beans such as the class under test – thus, eschewing the need for a JUnit4 **@Before** annotated method, in most cases

» Example: Injecting a Service Class for Testing

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = TestConfig.class)
public class OrderServiceTests {
  @Autowired
  OrderService service;

  @Test
  public void submitOrder0Test() {
    Order order = new Order(1L,100.0);
    Confirmation conf = service.submit(order);
    assertNotNull(conf);
  }
}
```

When a test class inherits from another class, any configuration resources defined with **@ContextConfiguration** in the parent class (or any ancestor class) are also considered. For cases where the test class should override the base definitions, set attribute **inheritLocations()=false**.

Table below summarizes the annotations provided by **Spring Test** to control tests setup and execution.

Annotation	Description
`@ContextConfiguration`	Define configuration resource for `ApplicationContext` (XML or Java)
`@WebAppConfiguration`	Denotes a webapp test using an `WebApplicationContext`
`@ActiveProfiles`	Define bean profile(s) to activate
`@TestPropertySource`	Load settings from a property file into the environment before test execution
`@DirtiesContext`	Mark test method as having side-effects – modifying beans state.
`@ContextHierarchy`	Define a hierarchy of `ApplicationContext`
`@TestExecutionListeners`	Install custom test listeners
`@BootstrapWith`	Tailor framework with complete set of custom strategies

Table below summarizes the annotations provided by **Spring Test** specifically to be used with **JUnit4**.

Annotation	Description
`@IfProfileValue`	Conditionally ignore test method
`@ProfileValueSourceConfiguration`	Alternative source of settings to be evaluated in `@IfProfileValue`
`@Timed`	Max. accepted time for valid test to run
`@Repeat`	Run test method N times

Test Classes with Multiple Test Methods

When test classes define more than one test method, the execution of the tests needs to be carefully considered since a test method that has side-effects may interfere with the behaviour of other test methods (e.g. by modifying the state of a stateful singleton bean). To mitigate for this, the annotation **@DirtiesContext** can be used in a test method to force Spring to recreate the **ApplicationContext**, rather than use a cached instance, while setting up the execution of the following test method(s).

» Example: Marking Test Methods for Side-Effects

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=TestConfig.class)
public class OrderServiceTests {
  @Autowired
  OrderService service;  // stateful bean

  @Test
  @DirtiesContext
  public void shopClosedTest() {
    service.setOpen(false);
    ...
  }
```

```
  @Test
  public void cancelOrderTest() { ... }
}
```

Testing with Profiles

When beans are defined to exist only in specific **Spring** *environment profiles*, those profiles should be enabled in order to run tests that use them. Annotation **@ActiveProfiles** provides a convenient (IDE/build-tool neutral) way to set the Spring environment variable **spring.profiles.active** and therefore enable the profiles for the test **ApplicationContext**.. Annotation **@ActiveProfiles** also supports attribute **inheritLocations()**. This should be set to t**rue** when profile activation of a child test class should overwrite, rather than add, the definitions in the parent class. For cases where profile selection needs to be done dynamically – based on some run-time condition (e.g. reachability of a service) – an **ActiveProfilesResolver** strategy can be set in attribute **resolver()**.

The annotation **@TestPropertySource** can be used when the settings of an application, captured in the Spring **Environment** implicit bean, need to be modified for the purpose of a test. The **value()** or **locations()** attribute specifies the location of a properties file – either in traditional key-value entries in a text file format, or in XML format. Setting can also be defined *inline* with attribute **properties()**. Setting defined with **@TestPropertySource** take precedence over settings defined in the base **Environment** (i.e. for same keys). Additionally, inlined settings take precedence over settings loaded from a property file. Annotation **@TestPropertySource** supports attribute **inheritLocations()** and **inheritProperties()** with the expected semantics.

» Example: Activating Profiles for Tests

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = TestConfig.class)
@ActiveProfiles({"dev", "jpa"})
public class OrderServiceTests { ...
}

@Profile("jpa")
public class JpaOrderRepository implements OrderRepository {…}

@Profile("dev")
@Bean DataSource dataSource() {
  return new EmbeddedDatabaseBuilder()….build();
}
```

» Example: Loading Test Properties into Environment

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = TestConfig.class)
@TestPropertySource(value="test.properties",
  properties = { "locale = en_UK", "port: 9090" })
public class OrderServiceTests { ...
}
```

Transactional Tests

When executing test methods that have side-effects on databases, possible interference between methods can not be avoided by simple use of **@DirtiesContext**. For these cases, three approaches are available:

- **Rollback Transactions** – Force the transactional context to always rollback, even if no **Exception** is thrown. The **@Transactional** annotation gives support for this approach.
- **SQL Scripts** – Run setup and cleanup SQL scripts before and after the test method is run. The **@Sql** annotation gives support for this approach.
- **"Brute Force"** – Delete table records, and repopulated database with test data. The **JdbcTemplate** class gives support for this approach.

The **@Transactional** annotation is the preferred approach to test transactional code, since it is more straightforward approach and might not be easy in many cases to write a SQL script to undo changes to database state. On the other hand, using SQL scripts is useful for populate and clean the database state, possibly in a way specific to each test method.

The **@Transactional** annotation can be used at the level of the method or at the level of the test class. In either case, it sets the default behaviour of the transaction to always rollback. This can be changed with annotation **@Commit**. If **@Commit** is used at the class level, annotation **@Rollback** can be used in individual methods to force a rollback.

As usually, a **PlatformTransationManager** should be configured as a Spring bean to run transactional code. It is assumed by default that its *bean name* is **transactionManager**. The **value()** or **transactionManager()** attributes should be set when set for an alternative name.

Transactional methods defined in Spring beans – usually in service-layer component annotated with stereotype **@Service** – will participate in the transactional context setup by the **Spring Test** framework. Notice, however, that if the transaction propagation rule is defined as **REQUIRES** in some of these methods, the *rollback-only* behaviour does not apply.

Method-level annotations **@BeforeTransaction** and **@AfterTransaction** define methods to run and after a transaction in run. This contrasts with **JUnit4** annotations **@Before** and **@After** that run while the transaction is still active.

The static utility methods in class **TestTransaction** also allows for programmatic control and demarcation of transactions (*since* Spring 4.1). Method **start()** and **end()** demark the boundaries of the transaction, and methods **flagForCommit()** and **flagForRollback()** set transactions for "normal" commit or rollback-only modes.

» Example: Testing with Rollback-only Transactions

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes= TestConfig.class)
public class OrderServiceTests {
    @Autowired
    OrderService service;

    @Test
    @Transactional
    public void submitOrdeTest() {
        Order order = new Order(1L,100.0);
        service.submit(order);
        service.issueInvoice(order);
    }

    @Test
    @Transactional
    public void listOrderTest() {
        Order order = new Order(1L,100.0);
        service.addOrder(order);
        assertEquals(1, service.count());
    }
}
```

» Example: Programmatic Control of Transaction in Test

```
@Test
public void orderSubmitTest() {
  TestTransaction.flagForCommit();
  TestTransaction.start();
  //...
  TestTransaction.end();
}
```

Execution of SQL Scripts in Tests

When running tests that use relational databases, a common requirement is to pre-populate the database with some data (usually made up) for the sole purpose of the tests. This can be done with a **ResourceDatabasePopulator** – e.g. globally when configuring a **DataSource,** or programmatically in test classes – e.g. using static utility methods defined in **ScriptUtils.**

It is also possible to define SQL scripts to run on a per test method basis declaratively with annotation **@Sql**. By default, scripts are run before a test method is run, but by setting the attribute **executionPhase()** is also possible to run the scripts to clean up the database state after a test method is run. The annotation **@SqlGroup** can be used to group multiple **@Sql** in a single method – if using <Java7. Annotation **@SqlConfig** can be optionally used to configure details how scripts are parsed and executed.

» Example: Populate & CleanUp Database w/ SQL Scripts

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes= TestConfig.class)
@SqlConfig(transactionMode = TransactionMode.ISOLATED,
errorMode=ErrorMode.IGNORE_FAILED_DROPS)
public class OrderServiceTests {
  @Autowired OrderService service;

  @SqlGroup({
    @Sql("insert-test-data.sql"),
    @Sql(scripts="delete-test-data.sql",
      executionPhase = ExecutionPhase.AFTER_TEST_METHOD)})
  @Test
  public void listOrdersTest() {
    List<Order> orders = service.findAllOrders();
    assertEquals(10, orders.size());
}
```

Table below summarizes the annotations provided by **Spring Test** to support test methods running in a transactional context and/or accessing relational databases.

Annotation	Description
`@Trasactional`	Start transactional context and always rollback
`@Commit`	Cancel the rollback-only setting
`@Rollback`	Cancel the @Commit setting
`@BeforeTransaction`	Execute method before transaction starts
`@AfterTransaction`	Execute method after transaction ends
`@Sql`	Execute SQL script
`@SqlConfig`	Config for SQL script parsing&processing
`@SqlGroup`	Group multiple @Sql (if < Java8)

Testing in a Web Environment

When testing web application the annotation **@WebAppConfiguration** should be used in test classes. This ensures that a **WebApplicationContext** is created to managed the spring beans with support for *session-scoped* and *request-scoped* beans. Several mock objects are also created to ensure that webapps can be tested with a completely defined environment, including a **MockServletContext** – cached and reused across test methods, unless the **@DirtiesContext** annotation is used. Several thread-local mock objects are also created per test method to emulate corresponding objects in a Servlet context, including: **MockHttpSession**, **MockHttpServletRequest**, **MockHttpServletResponse**, and Spring provided **ServletWebRequest**.

» Example: Test Class for Web Environment

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(classes=
  {WebConfig.class, TestConfig.class})
@ActiveProfiles({"web", "dev"})
public class OrderControllerTests {
  @Autowired WebApplicationContext context; // cached
  @Autowired MockServletContext servletContext; // cached
  @Autowired MockHttpSession session; // thread-local
  @Autowired ServletWebRequest webRequest;  // thread-local
}
```

» Example: Sample Controller & Test Method

```
@Controller
public class OrderController {
  @RequestMapping("/order/{id}")
  public ModelAndView show(@PathVariable Long id,
    ServletWebRequest webRequest) { … }
}
```

```
@Test
public void showTest() {
    Long id = 1L;
    ModelAndView modelAndView = controller.show(id, webRequest);
    assertNotNull(modelAndView);
    assertNotNull(modelAndView.getView());
    assertNotNull(modelAndView.getModel());
    assertEquals(1, modelAndView.getModel().size());
}
```

» Example: Testing with Session-Scoped Beans

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(classes = WebConfig.class)
public class ShoppingServiceTests {
  @Autowired MockHttpSession session;
  @Autowired ShoppingService service;
  @Autowired ShoppingCard card;

  @Test
  public void checkOutTest() {
    session.setAttribute("currency", "EUR");
    card.addItem(new Item(1, new Product("PAD-ABC")));
    Order order = service.checkOut();
    assertEquals(1, order.size());
  }
}
```

Testing with Context Hierarchies

In some application deployment modes and environments, more than one **ApplicationContext** is used, usually with a single *root context* and one or more *child contexts* in one or more levels (e.g. Spring MVC **DispatcherServlet** creates a child **ApplicationContext**, different from the a root one created by a **WebInitializer** or **ContextLoaderListener**). The annotation **@ContextHierarchy** allows this hierarchical organization to be recreated and emulated when running tests. The **value()** attribute takes an array of **@ContextConfiguration** annotations defining the configuration resources for different **ApplicationContext**. The **name()** attribute in each one can be used to selectively control overriding when class hierarchies are used.

» Example: Runnig Web Test w/ Context Hierarchy

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextHierarchy({
  @ContextConfiguration(name="root",classes=AppConfig.class),
  @ContextConfiguration(name="web", classes = WebConfig.class)
})
@ActiveProfiles({"web", "dev"})
public class OrderControllerTests {
  @Autowired WebApplicationContext wac; // child
}
```

Out-of-Container Web Testing

Spring Test framework support out-of-the-container testing of web apps and REST-WS build with **Spring MVC**. This is the ability to "deploy" applications in a virtual (mock) *Servlet Container* environment, allowing full integration tests of the web/REST layer to be performed – including the integration with **Spring MVC** without starting a container (standalone or embedded). The benefit of this is to allow for a more thoroughly testing, such as: data-binding, validation, message converters, etc. This is achieved trough a fluent API in class **MockMVC**, which allows emulation of HTTP requests without sending messages over a real transport-layer over the wire.

A **MockMVC** object is created by a factory method in class **MockMvcBuilders**. Factory method **webAppContextSetup()** takes an **ApplicationContext** as argument and creates a **MockMVC** that makes use of the full configuration of **Spring MVC** defined in the context. Factory method **standaloneSetup()** is used for simpler cases with just one (or a few) controller and simpler **Spring MVC** configuration.

Method **MockMVC.perform()** accepts as input a **RequestBuilder** object defining the details of the request. A fluent API can be convenient used to initialize the **RequestBuilder**, including: URL or path, including the use of path variables; HTTP method – e.g. **get()**, **post()**; request body – **content()**; and headers – e.g. **contentType()**, **accept()**. The result object **ResultActions** also supports a fluent API that allows expectations on results to be set with method **andExpect()** taking as input a **ResultMatcher**. Static utility methods provide a convenient way to defined the expectations using different types of **ResultMatcher**, such as: response status code – **status()**, content header and body – **content()**. For response body with portable formats (JSON and XML) there is also dedicated methods to test the response – **jsonPath()** and **xpath()**. Method **andReturn()** can be used to get direct access to the responses of the handler method in the invoked controller (e.g. model attributes, redirect URL, etc.).

» Examle: Testing a REST-WS Out-of-the-Container

```
import static org.sf.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.sf.test.web.servlet.result.MockMvcResultMatchers.*;
import static org.sf.test.web.servlet.request.MockMvcResultHandlers.*;
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(classes= WebConfig.class)
public class OrderControllerTests {
  @Autowired WebApplicationContext context;
  private MockMvc mockMvc;

  @Before
  public void setup() {
    mockMvc = MockMvcBuilders
      .webAppContextSetup(this.context).build();
  }

  @Test
  public void getOrder() throws Exception {
    mockMvc.perform(get("/order/{id}", 1)
      .accept(MediaType.parseMediaType("application/json")))
      .andDo(print())
      .andExpect(status().isOk())
      .andExpect(content().contentType("application/json"))
      .andExpect(jsonPath("$.status").value("OPEN"));
  }
}
```

Pseudo End-to-End Web Testing

Spring MVC Test framework also support integration with end-to-end testing frameworks, such as **HtmlUnit** (since 4.2). This allow more thoroughly testing of end-to-end behaviour, still without requiring deployment into a Servlet container (e.g. detailed testing of web page rendering – if using VDLs like **FreeMarker** or **Thymeleaf**, but not JSP), or execution of Javascript code (e.g. to test DOM changes). The **HtmlUnit** API main object **WebClient** is initialized with factory methods in **MockMvcWebClientBuilder**. Access to URLs in the **localhost** are served by **Spring MVC Test**, rather than going through a HTTP connection on the wire.

» Example: Testing with HtmlUnit & Spring MVC Test

```
private WebClient webClient;

@Before
public void setup() {
  this.webClient = MockMvcWebClientBuilder
    .webAppContextSetup(context).build();
}

@Test
public void editOrderTest() throws Exception {
  HtmlPage page  = webClient
    .getPage("http://localhost/order/1/edit");
  HtmlForm form = page.getHtmlElementById("orderForm");
  HtmlTextInput status = page.getHtmlElementById("status");
  status.setValueAttribute("CANCELED");
  HtmlSubmitInput submit = form
    .getOneHtmlElementByAttribute("input", "type", "submit");
  HtmlPage page2 = submit.click();
  assertEquals("http://localhost/order/1",
    page2.getUrl().toString());
}
```

Testing with Spring Boot

Spring Boot provides additional annotations and mechanisms to support integration testing. Table below summarizes some of these annotations.

Annotation	Description
`@SpringBootTest`	Alternative to @ContextConfiguraiton with additional Spring Boot setting
`@TestConfiguration`	Configuration class for Tests only
`@TestComponent`	Component class for Tests only
`@MockBean`	Create & Inject (Mockito) Mock object

Spring Security Testing

Spring Security also provides some annotations specifically to support integration testing in secured application. Most are used setup the **SecurityContext** with an authenticated user, in order to invoke and test protected methods. Table below summarizes some of these annotations.

Annotation	Description
`@WithMockUser`	Run with a mock User in **SecurityContext**
`@WithAnonymousUser`	Run test with anonymous User
`@WithUserDetails`	Run with user from **UserDetailsService**
`@WithSecurityContext`	Run with a mock **SecurityContext**

Extending Spring Test Framework

Spring Test framework is highly extensible to accommodate additional features on the execution of tests. The interface **TestExecutionListener** defines a set of callback methods that can be used to pre/post-process instances of test classes. They can be installed in individual test classes with annotation **@TestExecutionListeners** or globally by specifying the fully-qualified class name in file **META-INF/spring.factories**. The **@Order** annotation can be used to control the call order for the listeners. More detailed customization can be done with interface type **TestContextBootstrapper**, installed with annotation **@BootstrapWith**.

Resources

- Testing in *Spring Framework Reference Manual* – http://docs.spring.io/spring/docs/4.3.4.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#testing
- Testing in *Spring Boot Reference Manual* – http://docs.spring.io/spring-boot/docs/2.0.0.BUILD-SNAPSHOT/reference/htmlsingle/#boot-features-testing
- Testing in *Spring Security Reference Manual* – http://docs.spring.io/spring-security/site/docs/4.2.0.BUILD-SNAPSHOT/reference/htmlsingle/#test

About the Author

Jorge Simão is a software engineer and IT Trainer, with two decades long experience on education delivery both in academia and industry. Expert in a wide range of computing topics, he his an author, trainer, and director (Education & Consulting) at **EInnovator**. He holds a B.Sc., M.Sc., and Ph.D. in Computer Science and Engineering.