# Spring Boot Testing

# Spring TestContext Framework

# Testing with JUnit 5

# Testing with JUnit 5

- JUnit 5 is the default JUnit version from Spring Boot 2.2
- Components
  - **JUnit Platform**
    - For launching testing frameworks on the JVM with build tool and IDE support
  - **JUnit Jupiter**
    - An extension model for writing tests and extensions in JUnit 5
  - **JUnit Vintage**
    - A TestEngine for running JUnit 3 & 4 tests on the platform

# JUnit 5 Annotations

- JUnit 5 annotations
  - `@BeforeEach`
  - `@BeforeClass`
  - `@AfterEach`
  - `@AfterAll`
  - `@Disabled`
  - `@DisplayName`
  - `@Nested`
  - `@ParameterizedTest`

# Unit Testing (Without Spring)

- Unit Testing
  - Tests one unit of functionality
  - Keeps dependencies minimal
  - Isolated from the environment (including Spring)
  - Uses simplified alternatives for dependencies
    - Test doubles like Stubs and/or Mocks

# Integration Testing (With Spring)

- Integration Testing
  - Tests the interaction of multiple components/units working together
  - All should work individually first (unit tests show this)
- Tests application classes in context of their surrounding infrastructure
  - Out-of-container testing, no need to run-up full application server
  - Infrastructure may be "scaled down"
    - In-memory database instead of full database
- Test Containers
  - lightweight, throwaway instances of common databases or anything that runs in a Docker container

# Spring Support for Testing

- Spring has rich testing support
  - Based on `TestContext` framework
    - Defines an `ApplicationContext` for your tests
  - `@ContextConfiguration`
    - Defines the Spring configuration to use
- Packaged in a separate module with many gems
  - `spring-test.jar`

https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#testcontext-framework

https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#integration-testing

# `@ExtendWith` in JUnit 5

- JUnit 5 has extensible architecture via `@ExtendWith`
  - JUnit 5 supports multiple extensions at the same time
- Spring's extension point is the `SpringExtension` class

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes={SystemTestConfig.class})
public class CustomerServiceTests {
    …
}
```

# @ExtendWith in JUnit 5

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes={SystemTestConfig.class})
public class CustomerServiceTests {

    @Autowired
    private CustomerService customerService;

    @Test
    public void shouldUpdateCustomerSuccessfully() {
        CustomerChangeConfirmation conf = customerService.update(…);
        …
    }
}
```

# `@SpringJUnitConfig`

- `@SpringJUnitConfig` is a "composed" annotation that combines
  - `@ExtendWith(SpringExtension.class)` from JUnit 5
  - `@ContextConfiguration` from Spring

```
@SpringJUnitConfig(SystemTestConfig.class)
public class CustomerServiceTests {
    …
}
```

# Alternative Autowiring for Tests

```
@SpringJUnitConfig(SystemTestConfig.class)
public class CustomerServiceTests {

    // @Autowired
    // private CustomerService customerService;

    @Test
    public void shouldUpdateCustomerSuccessfully(@Autowired CustomerService customerService) {
        CustomerChangeConfirmation conf = customerService.update(…);
        …
    }
}
```

# Including Configuration as an Inner Class

```
@SpringJUnitConfig
public class JdbcCustomerRepoTest {

    @Test
    public void shouldUpdateDatabaseSuccessfully() {…}

    @Configuration
    @Import(SystemTestConfig.class)
    static class TestConfiguration {

        @Bean
        public DataSource dataSource() { … }
    }
}
```

# Multiple Test Methods

Most Spring Beans are stateless/immutable singletons, never modified during any test. No need for a new context for each test.

```
@SpringJUnitConfig(classes=SystemTestConfig.class)
public class CustomerServiceTests {

    @Autowired
    private CustomerService customerService;

    @Test
    public void successfulTransfer() { … }

    @Test
    public void failedTransfer() { … }
}
```

# @DirtiesContext

- Forces context to be closed at end of test method
- Next test gets a new Application Context
  - Cached context destroyed, new context cached instead

```
@Test
@DirtiesContext
public void testAddressLimitExceeded() {
    customerService.setMaxAddresses(0);
    // Add new address, expect a failure
 }
```

# Test Property Sources

- Custom properties just for testing
  - Specify one or more properties
    - Has higher precedence than sources
  - Specify location of one or more properties files to load
    - Defaults to looking for [classname].properties

```
@SpringJUnitConfig(SystemTestConfig.class)
@TestPropertySource(properties = { "username=foo", "password=bar" },
    locations = "classpath:/customer-test.properties")
public class CustomerServiceTests {
  // …
}
```

# Testing with Profiles

# Testing with Profiles

# Activating Profiles for a Test

- `@ActiveProfiles` inside the test class
  - Activate one or more profiles
  - Beans associated with that profile are instantiated, also any beans with no profile
- Example: Two profiles activated – **jdbc and local**

```
@SpringJUnitConfig(LocalConfig.class)
@ActiveProfiles( { "jdbc", "local" } )
public class CustomerServiceTests { … }
```

# Profiles Activation with JavaConfig

- `@Profile` on `@Configuration` class or any of its `@Bean` methods

```
@SpringJUnitConfig(LocalConfig.class)
@ActiveProfiles("jdbc")
public class CustomerServiceTests {…}
```

```
@Profile("jdbc")
@Configuration
public class LocalConfig {


    @Bean
    public … {…}
}
```

```
@Configuration
public class LocalConfig {

    @Bean
    @Profile("jdbc")
    public … {…}
}
```

# Profiles Activation with Annotations

- `@Profile` on a Component class

```
@SpringJUnitConfig(LocalConfig.class)
@ActiveProfiles("jdbc")
public class CustomerServiceTests {
    …
}
```

```
@Repository
@Profile("jdbc")
public class JdbcAccountRepository {
    …
}
```

# Testing with Databases

# @Transactional

# `@Transactional` within Integration Test

- Annotate test method (or class) with @Transactional
  - Runs test methods in a transaction
  - Transaction will be rolled back afterwards
    - No need to clean up your database after testing!

```
@SpringJUnitConfig(RewardsConfig.class)
public class RewardNetworkTest {

    @Test
    @Transactional
    public void testRewardAccountFor() {
        …
    }
}
```

# Controlling Transactional Tests

```
@Transactional
@SpringJUnitConfig(RewardsConfig.class)
public class RewardNetworkTest {

    @Test
    @Commit
    public void testRewardAccountFor() {
    … // Whatever happens here will be committed
    }
}
```

# Testing with Spring Boot

# Testing Setup

# Testing Setup

- Adding `spring-boot-starter-test` and getting many testing libraries as dependency with scope test:
  spring-boot-test, spring-boot-autoconfigure-test, json-path, junit, assertj-core, mockito-core, hamcrest-core, hamcrest-libarary, jsonassert, spring-core, spring-test, ...

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

# Testing with @DataJpaTest

# Setting up a JPA Repository & Entity

```
public interface CustomerRepository extends JpaRepository<Customer, Long>{

    public Customer findByName(String name);
}
```

```
@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Size(min = 3, max = 20)
    private String name;
    // constructors, getters and setters
}
```

# @DataJpaTest (1)

```
@DataJpaTest
public class CustomerRepositoryTest {

   @Autowired
   private TestEntityManager entityManager;

   @Autowired
   private CustomerRepository customerRepository;

   // write test cases here
}
```

# @DataJpaTest (2)

```
@Test
public void whenFindByName_thenReturnCustomer() {
    // given
    Customer carmen = new Customer("Carmen");
    entityManager.persist(carmen);
    entityManager.flush();

    // when
    Customer found = customerRepository.findByName(carmen.getName());

    // then
    assertThat(found.getName()).isEqualTo(carmen.getName());
}
```

# @DataJpaTest (3)

- Configuring in-memory database (H2, HSQLB, Derby)
- Configuring Hibernate, Spring Data, and the `DataSource`
- Performing `@EntityScan`
- Turning on SQL logging by default

# Mocking with @MockBean

# Setting up a Service

```
@Service
public class CustomerServiceImpl implements CustomerService{

    @Autowired // Field injection is evil ;)
    private CustomerRepository customerRepository;

    @Override
    public Customer getCustomerByName(String name){
        return customerRepository.findByName(name);
    }
}
```

# @MockBean (1)

```
@ExtendWith(SpringExtension.class)
public class CustomerServiceImplTest {

    @TestConfiguration
    static class ContextConfiguration{

        @Bean
        public CustomerService customerService() {
            return new CustomerServiceImpl();
        }
    }

    @Autowired
    private CustomerService customerService;

    @MockBean
    private CustomerRepository customerRepository;

    // write test cases here
}
```

# @MockBean (2)

```
@Test
public void whenValidName_thenCustomerShouldBeFound() {
    String name = "carmen";
    Customer carmen = new Customer(name);

    given(customerRepository.findByName(name)).willReturn(carmen);

    Customer found = customerService.getCustomerByName(name);

    assertThat(found.getName()).isEqualTo(name);

    verify(customerRepository).findByName(name);
}
```

# @MockBean (3)

- `@TestConfiguration` used on classes in `src/test/java`
  - Spring Boot annotation that does not prevent auto-detection of `@SpringBootConfiguration` like `@Configuration`
- `@MockBean` creates mocking proxy which can be configured for the test

# Testing with @WebMvcTest

# Setting up a Controller

```
@RestController
@RequestMapping("/api")
public class CustomerRestController {

    @Autowired // Field injection is evil ;)
    private CustomerService customerService;

    @GetMapping("/customers")
    public List<Customer> getAllCustomers() {
        return customerService.getAllCustomers();
    }
}
```

# @WebMvcTest (1)

```
@WebMvcTest(CustomerRestController.class)
public class CustomerRestControllerTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private CustomerService service;

    // write test cases here
}
```

# @WebMvcTest (2)

```
@Test
public void givenCustomers_whenGetCustomers_thenReturnJsonArray() throws Exception {

    Customer carmen = new Customer("carmen");

    List<Customer> allCustomers = Arrays.asList(carmen);

    given(service.getAllCustomers()).willReturn(allCustomers);

    mvc.perform(get("/api/customers")
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$",hasSize(1)))
        .andExpect(jsonPath("$[0].name", is(carmen.getName())));

    verify(service).getAllCustomers();
}
```

# @WebMvcTest (3)

- `@WebMvcTest` will be limited to bootstrap a single controller
- `@MockBean` to provide mock implementations for required dependencies
- `@WebMvcTest` also auto-configures MockMvc
- Easy testing MVC controllers without a full HTTP server
- JSONPath is used to parse and verify the JSON Response

# Integration Testing with @SpringBootTest

# @SpringBootTest (1)

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT, classes = Application.class)
@AutoConfigureMockMvc
@TestPropertySource(locations = "classpath:application-integrationtest.properties")
public class CustomerRestControllerIntTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private CustomerRepository repository;

    // write test cases here
}
```

# Setting up Properties

- `application-integrationtest.properties` contains details to configure persistence storage H2

```
# application-integrationtest.properties
spring.datasource.url=jdbc:h2:mem:test
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```

# @SpringBootTest (2)

```
@Test
public void givenCustomers_whenGetCustomers_thenStatus200() throws Exception {

    createTestCustomer("carmen");

    mvc.perform(get("/api/customers")
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(content()
        .contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$[0].name",is("carmen")));
}
```

# @SpringBootTest (3)

- Integrating different layers of the application
  - No mocking involved
- `@SpringBootTest` needs to start up a container for tests
  - Spring Boot and AutoConfiguration to the rescue
- `@TestPropertySource` will override existing properties
- MockMvc can be used for testing with real ports or with simulated infrastructure