# 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 EmployeeRepository extends
    JpaRepository<Employee, Long>{

    public Employee findByName(String name);
}


@Entity
public class Employee {

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

@Size(min = 3, max = 20)
private String name;

// constructors, getters and setters
}
```

# @DataJpaTest (1)

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class EmployeeRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private EmployeeRepository employeeRepository;

    // write test cases here

}
```

# @DataJpaTest (2)

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

    // when
    Employee found =
       employeeRepository.findByName(carmen.getName());

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

# @DataJpaTest (3)

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

# Mocking with @MockBean

# Setting up a Service

```
@Service
public class EmployeeServiceImpl implements EmployeeService{

    @Autowired
    private EmployeeRepository employeeRepository;


    @Override
    public Employee getEmployeeByName(String name){
        return employeeRepository.findByName(name);
    }
}
```

# @MockBean (1)

```
@RunWith(SpringRunner.class)
public class EmployeeServiceImplTest {

    @TestConfiguration
    static class ContextConfiguration{

        @Bean
        public EmployeeService employeeService() {
            return new EmployeeServiceImpl();
        }
    }

    @Autowired
    private EmployeeService employeeService;

    @MockBean
    private EmployeeRepository employeeRepository;

    // write test cases here
}
```

# @MockBean (2)

```
@Test
public void whenValidName_thenEmployeeShouldBeFound() {
    String name = "carmen";
    Employee carmen = new Employee(name);

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

    Employee found = employeeService.getEmployeeByName(name);

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

# @MockBean (3)

- @TestConfiguration used on classes in src/test/java
  - Not picked up by component scanner
- @MockBean creates mocking proxy which can be configured for the test

# Testing with @WebMvcTest

# Setting up a Controller

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

    @Autowired
    private EmployeeService employeeService;


    @GetMapping("/employees")
    public List<Employee> getAllEmployees() {
        return employeeService.getAllEmployees();
    }
}
```

# @WebMvcTest (1)

```
@RunWith(SpringRunner.class)
@WebMvcTest(EmployeeRestController.class)
public class EmployeeRestControllerTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private EmployeeService service;

    // write test cases here
}
```

# @WebMvcTest (2)

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

    Employee carmen = new Employee("carmen");

    List<Employee> allEmployees = Arrays.asList(carmen);

    given(service.getAllEmployees()).willReturn(allEmployees);

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

# @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)

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

public class EmployeeRestControllerIntTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private EmployeeRepository repository;

    // write test cases here
}
```

# Properties

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

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

# @SpringBootTest (2)

```
@Test
public void givenEmployees_whenGetEmployees_thenStatus200()
     throws Exception {

    createTestEmployee("carmen");

    mvc.perform(get("/api/employees")
      .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