Integration Testing

Testing, Integration Testing

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

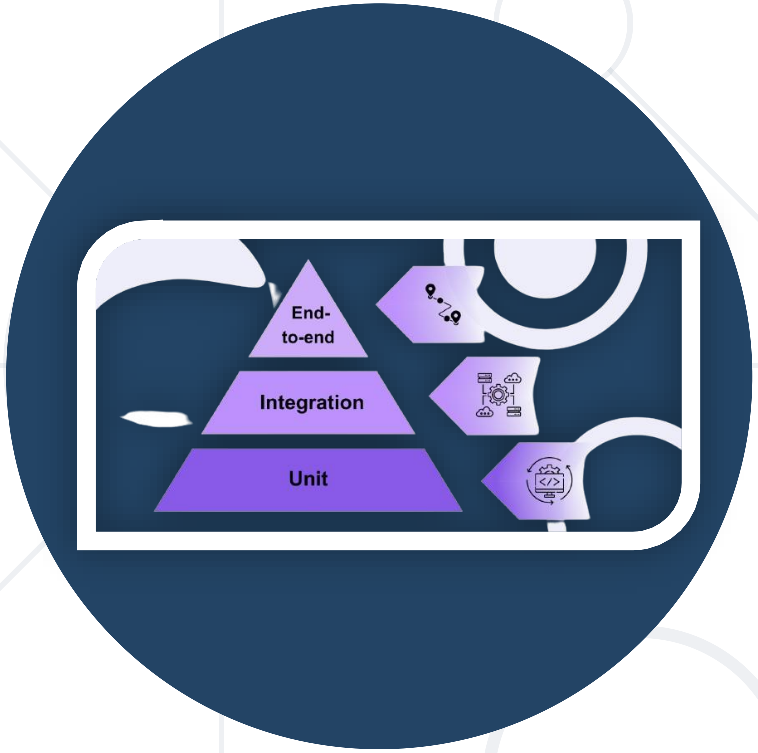
Have a Question?

Software University

sli.do

#java-web

Table of Contents

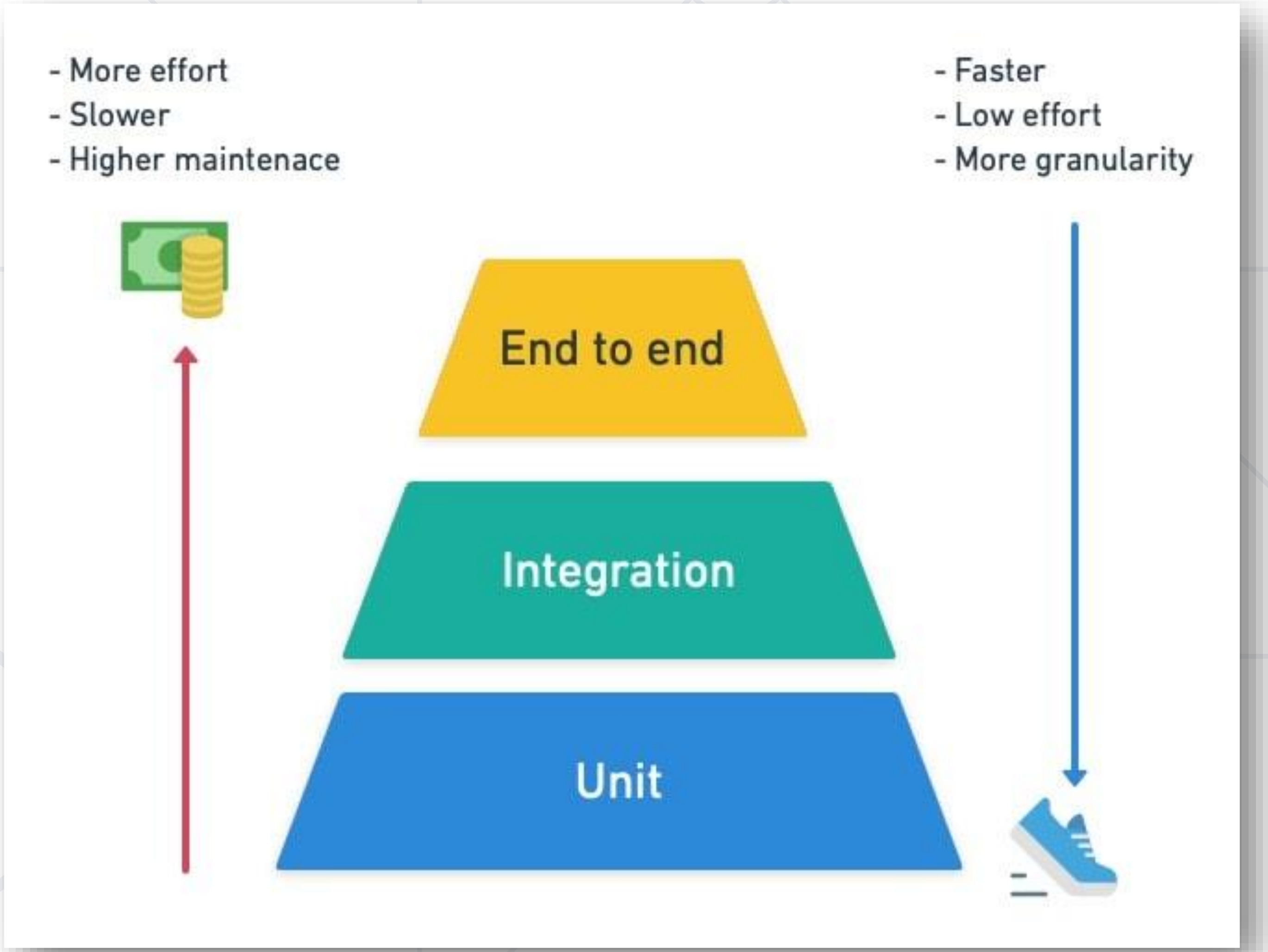


Testing

What Does "Testing Pyramid" Mean?

- The **Testing Pyramid** is a fundamental concept in software quality assurance
- It outlines a tripartite approach to testing: **unit tests** at the base, followed by foundational unit validation, followed by interconnected **integration** check-out, and culminating in all-encompassing **end-to-end** evaluations

The Testing Pyramid

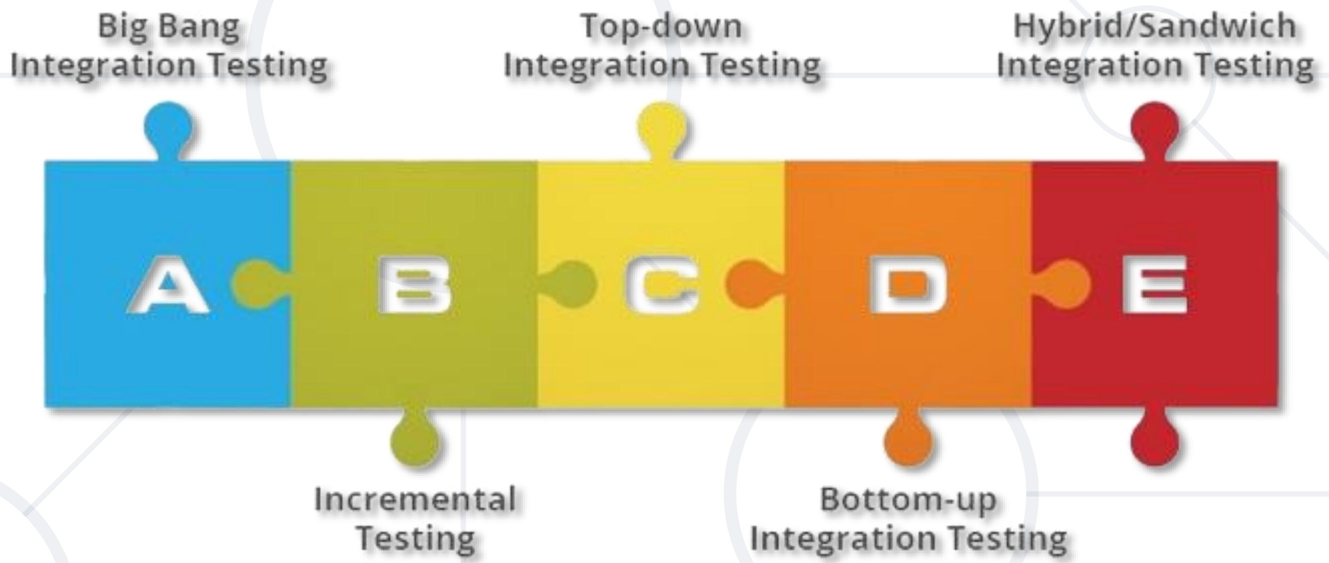


Integration Testing

Integration Testing

- **Integration testing** is a phase in software testing where **individual units** or **components** of the software are combined and tested as a **group**
- The **purpose** of integration testing is to **identify** issues that occur when these components interact with each other
- In Java, integration testing often involves testing interactions between different **classes**, **modules**, or **services**, including external systems like **databases**, **web services** or **third-party libraries**

Integration Testing



Importance of Integration Testing

- **Detects Interface Issues**: Ensures that different modules or services interact correctly with each other
- **Catches Integration Defects**: Identifies defects that might not be apparent in unit tests
- **Ensures End-to-End Functionality**: Verifies that the system works as a whole, ensuring end-to-end functionality
- **Validates Data Flow**: Confirms that data is correctly passed and processed across modules

Mocking

- Mocking in integration testing **involves simulating** the behavior of complex components or external systems that your system interacts with, to create a **controlled** and **predictable** environment
  - This allows you to test the **interactions** and **integration** of various components within your system without relying on actual external dependencies

Why Mocking in Integration Testing?

- **Isolate Test Environment** - Prevent tests from being dependent on external systems or services
- **Control and Predictability** - Provide predictable responses and behaviors for external dependencies
- **Speed** - Speed up tests by avoiding actual network calls or resource-heavy operations
- **Consistency** - Ensure tests are consistent and not affected by changes in external systems

Setting Up Mocking in Integration Testing

- Define the Controller

```java
@RestController
public class CurrencyController {

  private final ExRateService exRateService;

  public CurrencyController(ExRateService exRateService) {
    this.exRateService = exRateService;
  }
. . .
```

Setting Up Mocking in Integration Testing

```
. . .
 @GetMapping("/api/convert")
  public ResponseEntity<ConversionResultDTO> convert(
      @RequestParam("from") String from,
      @RequestParam("to") String to,
      @RequestParam("amount") BigDecimal amount
  ) {
    BigDecimal result = exRateService.convert(from, to, amount);

    return ResponseEntity.ok(new ConversionResultDTO(
        from,
        to,
        amount,
        result
    ));
  }
```

Setting Up Mocking in Integration Testing

- Configure the Test Class
  - We will use **@MockBean** to mock the ExRateService

```
@SpringBootTest
@AutoConfigureMockMvc
@ExtendWith(SpringExtension.class)
public class CurrencyControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private ExRateService exRateService;

...
```

Setting Up Mocking in Integration Testing

```
. . .
@Test
    public void testConvert() throws Exception {
        //Arrange
        BigDecimal mockResult = new BigDecimal("100.00");
        when(exRateService.convert(anyString(), anyString(),
any(BigDecimal.class))).thenReturn(mockResult);
//Act & Assert
        mockMvc.perform(get("/api/convert")
                .param("from", "USD")
                .param("to", "EUR")
                .param("amount", "50"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.from").value("USD"))
                .andExpect(jsonPath("$.to").value("EUR"))
                .andExpect(jsonPath("$.amount").value(50))
                .andExpect(jsonPath("$.result").value(100.00));
    }
...
```

Setting Up Mocking in Integration Testing

```
. . .
@Test
    public void testConvert_notFound() throws Exception {
        //Arrange
        when(exRateService.convert(anyString(), anyString(),
any(BigDecimal.class))).thenThrow(new ApiObjectNotFoundException("Currency pair
not found", "USD-EUR"));
        //Act & Assert
        mockMvc.perform(get("/api/convert")
                .param("from", "USD")
                .param("to", "EUR")
                .param("amount", "50"))
                .andExpect(status().isNotFound())
                .andExpect(jsonPath("$.code").value("NOT FOUND"))
                .andExpect(jsonPath("$.id").value("USD-EUR"));
    }
}
```

Setting Up Mocking with @WithMockUser

- Create a Secure Controller

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

    @GetMapping("/secure")
    public ResponseEntity<String> getSecureData() {
        return ResponseEntity.ok("This is secured data");
    }
}
```

Setting Up Mocking with @WithMockUser

- Write the Integration Test

```
@WebMvcTest(SecureController.class)
public class SecureControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    @WithMockUser(username = "user", roles = {"USER"})
    public void givenAuthRequestOnPrivateService_shouldSucceedWith200()
throws Exception {
        mockMvc.perform(get("/api/secure"))
                .andExpect(status().isOk())
                .andExpect(content().string("This is secured data"));
    }
. . .
```

Setting Up Mocking with @WithMockUser

```
. . .
@Test
    public void givenNoAuthRequestOnPrivateService_shouldFailWith401()
throws Exception {
        mockMvc.perform(get("/api/secure"))
                .andExpect(status().isUnauthorized());
    }
}
```

MockMvcRequestBuilders Methods

- **perform()**
  - This method is used to perform an HTTP request
  - It takes a **RequestBuilder** as an argument, which can be created using static methods from **MockMvcRequestBuilders**

```
mockMvc.perform(MockMvcRequestBuilders.get("/endpoint"));
```

MockMvcRequestBuilders Methods

- **andExpect()**
  - This method is used to **define expectations** on the result of the HTTP request
  - It is used to check the **status**, **headers**, and **content** of the response

```
mockMvc.perform(MockMvcRequestBuilders.get("/endpoint"))
        .andExpect(status().isOk())
        .andExpect(content().string("Expected response"));
```

MockMvcRequestBuilders Methods

- andDo()
  - This method allows you to perform **additional** actions with the result, such as logging or printing the response
  - It's often used with **print()** to output the response details

```
mockMvc.perform(MockMvcRequestBuilders.get("/endpoint"))
       .andDo(print());
```

MockMvcRequestBuilders Methods

- **andReturn()**
  - This method returns the **MvcResult** object, which can be used to inspect the result further
  - It's useful for more complex **verifications** or for extracting information from the response

```
MvcResult result =
mockMvc.perform(MockMvcRequestBuilders.get("/endpoint"))
                                      .andReturn();
```

Mocking in Integration Testing Summary

- **Mocking in Integration Testing** - Helps isolate the test environment, control responses, and improve test speed and consistency
- **RestTemplate with @MockBean** - Simple way to mock dependencies within Spring Boot applications

Basic Integration Test with @SpringBootTest

- Controller

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

    @Autowired
    private UserService userService;

    @GetMapping("/users/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        User user = userService.findById(id);
        return ResponseEntity.ok(user);
    }
}
```

Basic Integration Test with @SpringBootTest

- **Service**

```java
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public User findById(Long id) {
        return userRepository.findById(id).orElseThrow(() -> new
ResourceNotFoundException("User not found"));
    }
}
```

- **Repository**

```java
public interface UserRepository extends JpaRepository<User, Long> {
}
```

Basic Integration Test with @SpringBootTest

- **Integration Test**

```
@SpringBootTest
@AutoConfigureMockMvc
public class UserControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private UserRepository userRepository;

    @BeforeEach
    public void setUp() {
        userRepository.save(new User(1L, "John Doe"));
    }
. . .
```

Basic Integration Test with @SpringBootTest

```
. . .
    @AfterEach
    public void tearDown() {
        userRepository.deleteAll();
    }

    @Test
    public void testGetUserById() throws Exception {
        mockMvc.perform(get("/api/users/1"))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.name").value("John Doe"));
    }
}
```

Using @DataJpaTest for Repository Testing

- Repository Test

```
@DataJpaTest
public class UserRepositoryIntegrationTest {
    @Autowired
    private TestEntityManager entityManager;
    @Autowired
    private UserRepository userRepository;
    @Test
    public void testFindById() {
        User user = new User(1L, "John Doe");
        entityManager.persist(user);
        entityManager.flush();

        User found = userRepository.findById(user.getId()).orElse(null);
        assertThat(found.getName()).isEqualTo(user.getName());
    }
}
```

Integration Testing with @Testcontainers

- **Integration Test**

```
@SpringBootTest
@AutoConfigureMockMvc
@Testcontainers
public class UserControllerIntegrationTest {

    @Container
    public static MySQLContainer<?> mysql = new MySQLContainer<>("mysql:8.0.23")
        .withDatabaseName("testdb")
        .withUsername("test")
        .withPassword("test");

    @DynamicPropertySource
    public static void dynamicProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", mysql::getJdbcUrl);
        registry.add("spring.datasource.username", mysql::getUsername);
        registry.add("spring.datasource.password", mysql::getPassword);
    }
. . .
```

Integration Testing with @Testcontainers

```java
. . .
@Autowired
    private MockMvc mockMvc;
    @Autowired
    private UserRepository userRepository;
    @BeforeEach
    public void setUp() {
        userRepository.save(new User(1L, "John Doe"));
    }
    @AfterEach
    public void tearDown() {
        userRepository.deleteAll();
    }
    @Test
    public void testGetUserById() throws Exception {
        mockMvc.perform(get("/api/users/1"))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.name").value("John Doe"));
    }
}
```

Best Practices

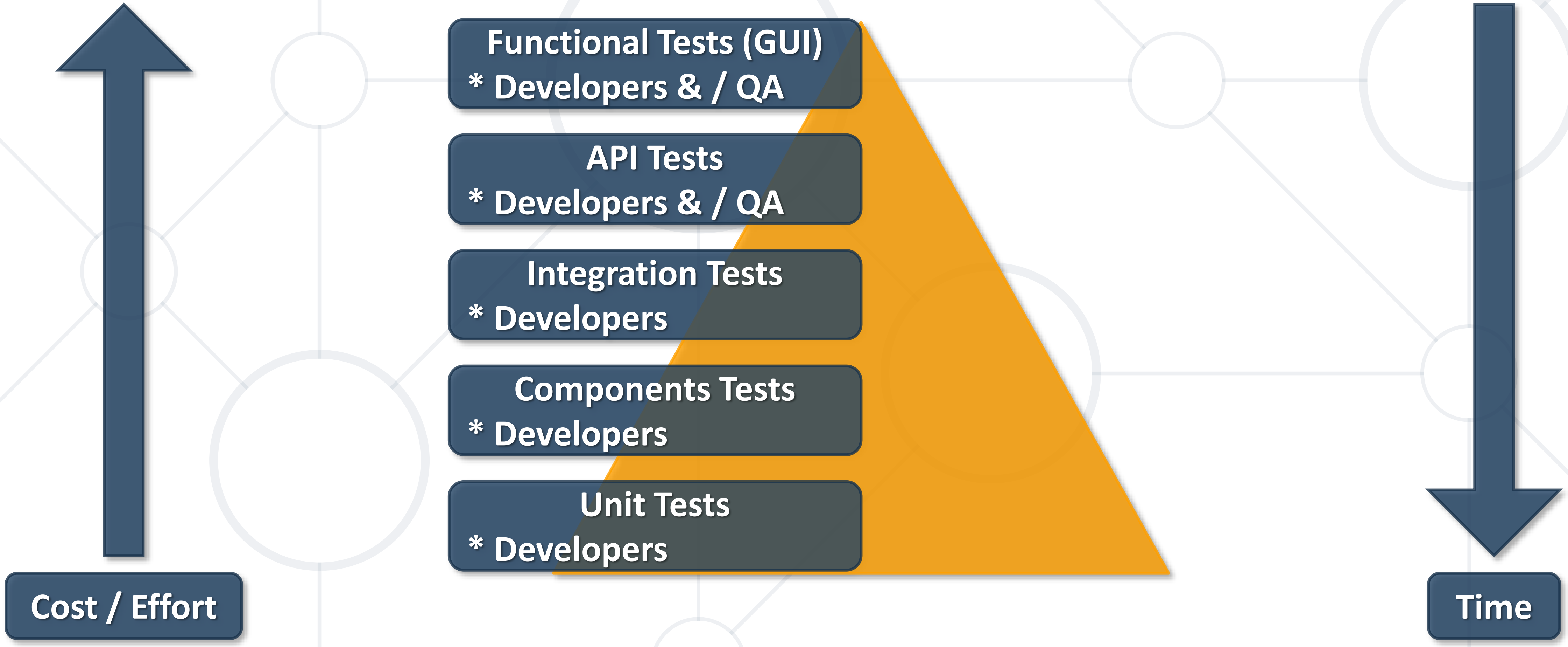
- **Isolation** - Ensure tests are independent and isolated to avoid interference
- **Setup and Teardown** - Use **@BeforeEach** and **@AfterEach** to set up and clean up the test environment
- **Data Management** - Use **in-memory** databases or reset the database state between tests to maintain consistency
- **Realistic Scenarios** - Use **Testcontainers** for more realistic integration testing, particularly for database-dependent tests

Conclusion

- Integration testing in Spring Boot is **essential** for verifying the correct interaction between **different** parts of an application
- By using annotations like **@SpringBootTest**, **@DataJpaTest**, and tools like **MockMVC** and **Testcontainers**, developers can ensure their applications are robust and function correctly in a **production-like** environment

Different Testing levels

- Different Testing levels **require different** time and resources



Summary

- **Testing** is an important part of the application lifecycle
  - New features need to be verified, before delivered to the clients
- **Integration Testing**
  - Testing the interactions between multiple layers of an application
  - The purpose is is to verify that different components of an application work together correctly

Questions?

Software University

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

СофтУни БУДИТЕЛ

Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank
Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR
Serving the high achievers

SOFTWARE GROUP

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License