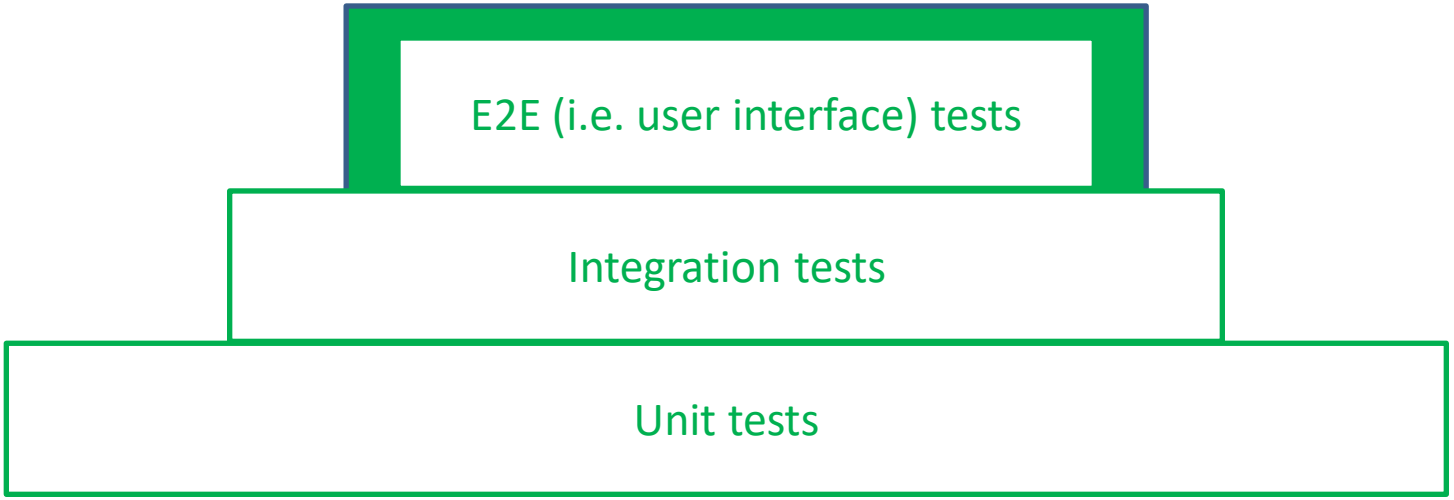
Test automation

Agenda

- Test automation overview
- API test automation
- GUI test automation

Software testing

The software testing pyramid



Test automation overview

Test automation overview

- The main purpose of test automation is to automate the process of manual testing
- Typically that is achieved by means of third-party tools and libraries
- Some of this libraries and tools can be used through standard unit testing frameworks like JUnit and TestNG

Test automation overview

- Although there are different strategies for test automation the two most widely adopted approaches are:
  - user interface test automation: various tools provide the possibility to perform actions and validations on the user interface of an application
  - API test automation: other tools provide the option to call the public API (i.e. REST or other form of HTTP API) of the application avoiding interaction with the user interface (if present)

Test automation overview

- To make the manual testing process automatable the application needs to follow certain guidelines in most cases

- If that is not the case any modifications to the application may result in an excessive effort to maintain the set of automation tests

For example using a browser-based test automation framework requires that DOM elements (like input fields or buttons) be accessible via a unique ID or other easy-to-use element accessor otherwise complex expressions (i.e. using XPath) need to be written that is not robust and easy to maintain

API test automation

API test automation

- API test automation is essential for applications that provide a programming interface to the outside world (i.e. in the from of REST or other HTTP endpoints)

- API test automation can also be used in combination with GUI test frameworks to exercise certain logic on the application that is available through endpoints but not from the GUI

API test automation

- The type of API tests depends on whether we have a running application against which to execute to API test:
  - If there is a running instance we can simply use an HTTP client (i.e. Apache HttpComponents Client or Jersey Client) to execute HTTP requests from JUnit or TestNG tests
  - If the test framework needs to start an instance of the application then there is typically more effort required to setup the application and run it as part of the test execution

Test against a running application

- The type of API tests depends on whether we have a running application against which to execute to API test:
  - If there is a running instance we can simply use an HTTP client (i.e. Apache HttpComponents Client or Jersey Client) to execute HTTP requests from JUnit or TestNG tests
  - If the test framework needs to start an instance of the application then there is typically more effort required to setup the application and run it as part of the test execution

Test against a running application

```
@Test
public void shouldAddDeviceApiTest() {
	RestTemplate restTemplate =
		new RestTemplate();
	MultiValueMap<String, Object> body =
		new LinkedMultiValueMap<String, Object>();
	body.add("param", "value");
	ResponseEntity<String> response =
		restTemplate.postForEntity(
			"http://localhost:8080/endpoint",
			body, String.class);
	System.out.println("Response is " +
		response.getBody());
}
```

Test against a running application

- A more declarative style of testing and validating result against an API endpoint is provided by the RestAssured library

```
@Test
public void shouldAddDeviceApiTest() {
        given()
                .param("param", “value")
        .when()
                .post("/endpoint")
        .then()
                .body(containsString(“text"));
}
```

Test running the application

- In some cases it is necessary to run the application as part of the test
- This can be considered an integration test
- Some frameworks like Spring provide capabilities to start the application with all of its dependencies as part of the test

Spring framework provides a Spring test framework with additional capabilities for unit and integration testing of Spring applications

Test running the application

- Spring framework provides a @SpringBootTest annotation for testing Spring boot applications using JUnit 5

- It wraps the SpringExtension class that can be registered with JUnit @ExtendWith annotation

- Allows for dependency injection of Spring beans in the unit test

Test running the application

```
@SpringBootTest
public class SomeControllerTest {

	@Autowired
	private SomeController controller;

	@Test
	public void shouldLoadController()
		throws Exception {
		assertNotNull(controller);
	}
}
```

Test running the application

```
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class RestTemplateTest {

	@Autowired
	private TestRestTemplate restTemplate;

	@LocalServerPort
	private int port;

	@Test
	public void shouldContainId() throws Exception {
		MultiValueMap<String, Object> body = new
			LinkedMultiValueMap<String, Object>();
		body.add("param", “value");
		assertTrue(this.restTemplate.postForObject(
			"http://localhost:" + port +
			"/endpoint", body, String.class)
			.contains("id"));
	}
}
```

Test running the application

- Instead of starting the Spring Boot application along with the HTTP server the web context can be mocked
- Spring test provides a @MockMvc annotation that allows for the execution of requests against the controllers using fake MVC objects (like requests and responses)
- Avoids the cost of starting a full web server as part of tests but mimics closely the same behavior

Test running the application

```
@SpringBootTest
@AutoConfigureMockMvc
public class MockMvcTest {

	@Autowired
	private MockMvc mockMvc;

	@Test
	public void shouldReturnDefaultMessage()
		throws Exception {
		this.mockMvc.perform(
			get("/endpoint"))
			.andDo(print())
			.andExpect(status().isOk())
			.andExpect(content()
			.string("some text"));
	}
}
```

Test running the application

```
@WebMvcTest(SomeController.class)
public class MockMvcTest {

	@Autowired
	private MockMvc mockMvc;

	@Test
	public void shouldReturnDefaultMessage()
		throws Exception {
		this.mockMvc.perform(
			get("/endpoint"))
			.andDo(print())
			.andExpect(status().isOk())
			.andExpect(content()
			.string("some text"));
	}
}
```

Test running the application

- Services used by a Spring bean (i.e. a controller) can be mocked using the @MockBean annotation that uses Mockito to mocking the service

```
@WebMvcTest(SomeController.class)
public class MockMvcTest {

        @Autowired
        private MockMvc mockMvc;

        @MockBean
        private SomeService service;

        …
{
```

GUI test automation

GUI test automation

- Different frameworks for GUI testing exist based on the type of GUI application (i.e. Swing, native Android or web-based)

- Since the majority of modern applications have browser-based frontend one of the most popular frameworks for web UI automation is Selenium

- Selenium provides features for automating the user interaction (such as a button click or input) with the web application

GUI test automation

- In some cases web elements are not easily accessible and one must use a different strategy (i.e. access a programming API) to achieve some of the test targets
- Other frameworks like SikuliX provide the possibility to recognize components in GUI using image recognition (trying to match screenshots against the GUI)
- Such frameworks can also be combined with frameworks like Selenium and minimize some of the limitations imposed by the GUI components of the application

Selenium

- Selenium provides an IDE for creating a running tests even without a programming language

- Client libraries are provided for a number of languages (including Java)

- Selenium provides a common API for commands that can be sent to a web browser using a browser-specific driver application

Earlier versions of Selenium were requiring a Java server called Selenium RC (remote control) to be started for the execution of the Selenium tests
Earlier versions of Selenium were also relying more on JavaScript rather than native browser capabilities like Selenium WebDriver

Selenium

```
@Test
public void shouldContainGoogleResult()
        throws InterruptedException {
        WebDriver driver = new ChromeDriver();
        WebDriverWait wait = new WebDriverWait(driver, 10);
        try {
                driver.get("https://google.com/ncr");
        driver.findElement(By.name("q"))
                .sendKeys("martin-toshev.com" + Keys.ENTER);
                String resultXpath =
                        "//a[@href='https://martin-toshev.org/']";
                WebElement firstResult = wait.until(
                        presenceOfElementLocated(
                                By.xpath(resultXpath)));
                firstResult.click();
        } finally {
                driver.quit();
        }
}
```

SikuliX

```
@Test
public void shouldContainGoogleResult()
        throws InterruptedException {
        WebDriver driver = new ChromeDriver();
        driver.get("https://google.com/ncr");
        driver.findElement(By.name("q"))
                .sendKeys("martin-toshev.com");
        Thread.sleep(5000);
        Screen screen = new Screen();
                try {
                        screen.click("google_search.png");
                        screen.click();
                } catch (FindFailed e) {
                        fail();
                }
}
```

Questions ?