Test Like a Charm
Docker Test Containers

Abstract

This is a story that most of us live every day. We all do software development that we think works like a charm. The Unit Test is part of every cycle in development and sometimes we do Integration Testing, and maybe automation testing. We all do Unit Test since it is most simple and required step of verification over implementation. Integration testing is also crucial part of cycle to verify, that implementation do what defines requirement or technical specification. Since those tests should verify communication and process flow between different services, which could involve many protocols of communication, it will be required to do setup locally, install database, some message broker and many other systems. Too much for local development environment. Then here comes automation testing as crucial part of continues delivery, where code is compiled if necessary and then packaged by a build server every time a change is committed to a source control repository, then tested by a number of different techniques. There are many tools out there which can help us do all of this quick and easy as much as possible. We can do it in Docker Test Containers, which we will see is quick to setup and run

Acceptance Testing Tools and Frameworks

There are many automation testing tools and frameworks on the market, and all of them have a many pros and cons to use it. We can shrink the list to the most popular testing tools and frameworks: FitNesse, Cucumber, Selenium WebDriver and SoapUI. The only common thing to all is that are tools for testing. With their own specific features for implementing tests and deployment, and focus on tests.
I will group them to the following groups: BDD, Web based application testing and Web Service functional testing.

FitNesse

This is a collaboration framework and widely accepted acceptance testing framework. It is an open source and has its own web server for the wiki, so it is easy to document the software, and

then run the tests from UI in order to verify the documentation against the software. It is easy for business people to define and run test scenarios, since the test are defined on a business level.
Specification can be in wiki syntax or text editor. However, the implementation of test methods is a class, called fixture, supports java, but also other languages.
The idea behind collaboration on wiki is to improve mutual understanding between testers and developers of the system and requirements.
Tests are described in FitNesse as coupling of inputs vs expected outputs in a decision table. Tests are expressed in tables with WikiWiki language.

```
|eg.Division|
|numerator|denominator|quotient?|
|10 |2 |5 |
|12.6 |3 |4.2 |
|100 |4 |33 |
```

The tool is Java based and shipped as executable jar, which includes a wiki engine, an embedded web server, testing engine and resources to create a web site in FitNesse style.
To test any sort of software we need to use fixtures. In order to test web service we need *RestFixture* or we could use *XmlHettTest*, also it is an option to implement some *Fixture* as we need.
Pros
Reusable fixtures between tests.
We can add or edit tests via GUI web application
Cons
We need to express test cases in WikiWiki language; I am not sure how complex the test scenarios could be.

Cucumber

A tool that supports Behaviour-Driven Development. It is an open source. Developers creates specification as a text and business approve. Specification consists of multiple examples or scenarios. Each scenario is a list of steps.

```
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday
  Scenario: Sunday isn't Friday
    Given today is Sunday
    When I ask whether it's Friday yet
    Then I should be told "Nope"
```

Cucumber reads executable specification and verifies that the software conforms to specification, and then generates report.
The syntax of scenarios is Gherkin Domain Specific Language. The steps can be written in java script or Java.

```
@Given("today is Sunday")
public void today_is_Sunday() {
```

```
    // Write code here that turns the phrase above into concrete actions
    throw new cucumber.api.PendingException();
}
```

We implement Assertions in those steps. What will be tested depends on implementation and what other tools will be used by implementation. Therefore, we could integrate with other test tools. One example for Rest API test is described here https://www.baeldung.com/cucumber-rest-api-testing.

Selenium WebDriver

Selenium is the most popular tool for automation testing of web application. It is an open source. Selenium is just used for emulating user-actions. Selenium does not necessarily do any validating automatically (other than page loads) however, that is up to implementation of tests.
WebDriver API is object oriented API.
There is Selenium IDE that is prototyping tool for building test scripts. Scripts are recorded in *Selenese*, a special test scripting language for Selenium. *Selenese* provides commands for performing actions in a browser (click a link, select an option) and for retrieving data from the resulting pages.
In order to implement Web Service test we would need to integrate with some other tools for automation of services like SoapUI, JMeter, Unirest …
Following are some examples for integration with Web Service API testing tools:
https://www.vinsguru.com/selenium-webdriver-how-to-test-rest-api/
https://www.baeldung.com/java-selenium-with-junit-and-testng
https://www.softwaretestinghelp.com/integrating-jmeter-selenium/
https://www.guru99.com/using-soapui-selenium.html

Tests can be written to any popular programming language.

SoaupUI

SoapUI is most popular testing tool for API's, which simplifies the process of creating automated end-to-end test scenarios across multiple layers (mobile, rest APIs, SOAP, Microservices, database, Web UIs) It is well documented. It has SoapUI IDE, support many protocols and types of messages (JMS, JSON, HTML, XML). Test creation is rapid and scriptless. Automatically generate test from a service definition. We could explore web services, do load testing, security testing, jms service testing, http web service testing, soap mocking, rest mocking, data driven tests.
SoapUI includes a Maven (version 2 and above) plugin for running SoapUI tests and mocks during a Maven build.
https://www.soapui.org/test-automation/maven/maven-2-x.html
It has free version and Pro version.

SoapUI and Seleneium WebDriver are complementary. Both tools can be integrated as described in details in following articles
https://www.guru99.com/using-soapui-selenium.html
https://webkul.com/blog/integrate-soapui-with-selenium/

Docker test containers

We could use docker test containers within our tests, we could write self-contained integration tests that depend on external resources. We just need docker image for database, web browser or message queues.
Executable is a jar file and separate jar files for each module.
In order to use any of containers in test we just need to define container rule in our test, most common is generic container, but also there are rule definitions with specialized functionality. They are for containers of common databases like MySQL, PostgreSQL; and others like web clients. In addition, there is an option for more complex services, we just need to define them in docker-compose file, so we can use *DockerComposeContainer* rule.
This is one example how to run test containers https://www.baeldung.com/docker-test-containers
To run SoapUI tests in docker we should follow these instructions https://support.smartbear.com/readyapi/docs/integrations/docker/soapui.html

Docker Test Containers in Action

What we would need to do before running Docker Test Containers?

We need to be sure we can run Docker containers so for local development environment we would need to install it, having on mind general Docker requirements, https://www.testcontainers.org/supported_docker_environment/

Since we are talking about tests, we would need supported JVM testing framework like Junit.

We could resolve all dependences with Maven/Gradle.

That is it.

How could help us with testing?

We can run everything in a container and I like it we have testcontainer image almost for everything we need in every day doing tests in development. To test Data Access layer we could use containerized instance of MySLQ, PosrtgeSQL or Oracle database and many other, without complex setup on development machine and each test runs in predefined state as we define per test case. Also, we could run web servers in container in order to run application integration

tests. Selenium browser based tests could run in fresh instance without browser state or plugins.

Since all containers are based on GenericContainer, we could do our customized testcontainer modules.

You noticed I started using testcontainer, which is a module already predefined and can be found on git-hub https://github.com/testcontainers

There are many modules out there we can use it with minimal setup.

TestContainers Demo

In a simple real case scenario, where we implement some spring boot application, with backend like PostgreSQL for database, by default we could start testing with in-memory database like H2. This is most likely first choice in development.

testcontainers-demo is simple spring boot application, in which there is only one entity EventLog, which is a table to store events and their status. At first it is defined like

```
@Entity(name = "eventlog")
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class EventLog {
    @Id
    private long id;
    private String type;
    private String status;

}
```

And one simple crud repository

```
@Repository
@Transactional
public interface EventLogRepository extends CrudRepository<EventLog, Long> {

}
```

Schema is defined in Liquibase changelog.

And we have dependence to H2 in order to run test

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.199</version>
    <scope>test</scope>
</dependency>
```

The test is pretty simple, initiate the state with one sql insert, and do test of context and one save of record.

```
@RunWith(SpringJUnit4ClassRunner.class)
@Sql(executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD, scripts =
"classpath:dao/TestData.sql")
public class ApplicationTests extends DBBaseIT {
```

```
    @Autowired
    private EventLogRepository eEventLogRepository;

    @Test
    public void testContext() {
        Assert.assertTrue(eEventLogRepository.findEventLogByType("Training") !=
null);
    }

    @Test
    @Transactional
    public void simpleEntity() {

        eEventLogRepository.save(EventLog.builder()
                .id(new Random().nextInt())
                .type("Presentation")
                .status("IN_PROGRESS")
                .build());

        Assert.assertTrue(eEventLogRepository.findEventLogByType("Presentation") !=
null);
    }
}
```

You can notice DBBaseIT abstract for integration test in which we could load context and configuration for specific back end. But for simple scenario like this there is no need to define any other configuration. The test is executed over h2 in-memory database.

Execution of this simple test took like 271ms on my local machine.

But at some point, we could face with an issue to test some functionality which depends on ProstgreSQL database. Then, you will need to setup PostgreSQL instance locally or you could do with embedded PostgreSQL instance. Which is not so hard to integrate with your tests. You could have and instance per test with expected state of database per test case. With this option, you will still have a local instance that will write the content to the file system on the host machine.

Since it is an event which could come from external system, maybe push notifications, in json format, we would like to keep original content of the event, raw json. Then we could add this to the entity which will look like this.

```
@Entity(name = "eventlog")
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
@TypeDefs({@TypeDef(name = "json", typeClass = JsonStringType.class),
        @TypeDef(name = "jsonb", typeClass = JsonBinaryType.class)})
public class EventLog {
    @Id
    private long id;
```

```
    private String type;
    private String status;

    @Type(type = "jsonb")
    @Column(name = "raw_json", columnDefinition = "jsonb")
    @JsonField
    private String rawJson;

}
```

In test we will add simpe test case.

```
@Test
@Transactional
public void entityWithJsonFieldType() {

    String json = "{\"type\": \"Presentation\", \"events\": [\"Test Containers
Demo\"], \"status\": \"Presentation\"}";

    eEventLogRepository.save(EventLog.builder()
            .id(new Random().nextInt())
            .type("Presentation")
            .status("IN_PROGRESS")
            .rawJson(json)
            .build());

    Assert.assertTrue(eEventLogRepository.findEventLogByType("TestName") != null);
}
```

Now in order to run the test we need to have integrated application with postgress embedded database. So for that we will create one module pg-embedded which will be spring-boot-starter-data-jpa integrated with otj-pg-embedded. Maven dependences will look like

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <!-- <scope>test</scope> -->
    </dependency>
    <dependency>
        <groupId>com.opentable.components</groupId>
        <artifactId>otj-pg-embedded</artifactId>
        <version>${pg-embedded.version}</version>
    </dependency>
    <dependency>
        <groupId>com.seavus.testcontainers</groupId>
        <artifactId>integration-testing</artifactId>
        <version>${project.version}</version>
```

```
    </dependency>
</dependencies>
```

In the configuration class, we will define instance of data source according to the properties for connection to database given in property file.

```
@Profile(EmbeddedPGConfig.EMBEDDED_PG)
@EnableAutoConfiguration
@Configuration
@PropertySource("classpath:/application-embedded-pg.yml")
public class EmbeddedPGConfig {

    public static final String EMBEDDED_PG = "embedded-pg";

    @Value("${spring.datasource.port}")
    Integer dbPort;

    @Bean("dataSource")
    DataSource getDataSource(EmbeddedPostgres embeddedPostgres) {
        return embeddedPostgres.getPostgresDatabase();
    }

    @Bean
    EmbeddedPostgres getEmbeddedPostgres() throws IOException {
        return EmbeddedPostgres.builder().setPort(dbPort).start();
    }
}
```

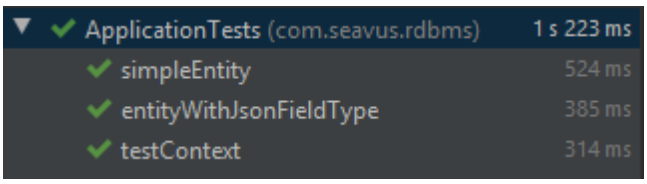
As you see in the implementation of the configuration, we get data source from the started instance of the embedded postgress.

Now it is enough just to switch in DBBaseIT to this profile and context.

```
@ContextConfiguration(classes = {EmbeddedPGConfig.class})
@ActiveProfiles({EmbeddedPGConfig.EMBEDDED_PG})
public class DBBaseIT extends BaseIT {

}
```

The execution of the test cases takes like



That could be a problem when you deploy application in the cloud. Today most of cloud solutions runs Docker instances managed by Kubernetes.

So, Docker is a set of PaaS products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files, they communicate with each other through well-defined channels. The software that hosts the containers is called Docker Engine.

Kubernetes is an open-source container-orchestration system for automating application deployment, scaling, and management. It was originally designed by Google, and is now maintained by the Cloud Native Computing Foundation. It aims to provide a "platform for automating deployment, scaling, and operations of application containers across clusters of hosts". It works with a range of container tools, including Docker.

The best option would be to have a database in their own container, which can have known state for each test case. Here comes PostgreSQL test container.

In order to integrate with PostgreSQL test container, we could do the similar thing. We will create new module testcontainer-pg in which we will define configuration for the container and data source.

This will be also spring-boot-starter-data-jpa but integrated now with postgresql testcontainer. Maven dependences will look like

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql </artifactId>
    <version>1.12.2</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.1</version>
</dependency>
```

```
<dependency>
    <groupId>com.seavus.testcontainers</groupId>
    <artifactId>integration-testing</artifactId>
    <version>${project.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
</dependency>
```

In the configuration class, we will define instance of data source according to the properties for connection to database given in property file.

```
@Data
@Profile(TestContainerPGConfig.TESTCONTAINER_PG)
@Configuration
@ConfigurationProperties(prefix = "spring")
@PropertySource("classpath:/application-testcontainer-pg.yml")
public class TestContainerPGConfig {

    public static final String TESTCONTAINER_PG = "testcontainer-pg";

    @Value("${spring.datasource.dbName}")
    String dbName;

    @Value("${spring.datasource.username}")
    String dbUsername;

    @Value("${spring.datasource.password}")
    String dbPassword;

    @Value("${spring.datasource.driverClassName}")
    String driverClassName;

    @Bean("dataSource")
    public DataSource dataSource(PostgreSQLContainer postgreSQLContainer) throws
SQLException {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(driverClassName);
        ds.setUrl(format("jdbc:postgresql://%s:%s/%s",
postgreSQLContainer.getContainerIpAddress(),

postgreSQLContainer.getMappedPort(PostgreSQLContainer.POSTGRESQL_PORT),
                postgreSQLContainer.getDatabaseName()));
        ds.setUsername(postgreSQLContainer.getUsername());
        ds.setPassword(postgreSQLContainer.getPassword());
        return ds;
    }

    @Bean
    public PostgreSQLContainer getPostgreSQLContainer() {
        PostgreSQLContainer postgres = (PostgreSQLContainer) new
```

```
PostgreSQLContainer("postgres:10.3").withDatabaseName(
                dbName).withUsername(dbUsername).withPassword(dbPassword);
        // .withStartupTimeout(Duration.ofSeconds(600));
        postgres.start();
        return postgres;
    }
}
```

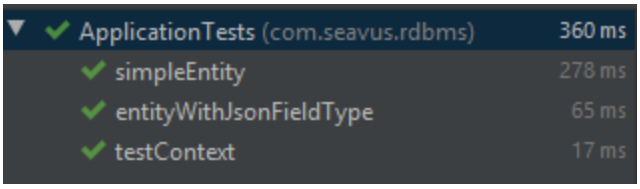
As you may see in the implementation of the configuration, we get data source from started PostgreSQL container instance.

Now it is enough just to switch in DBBaseIT to this profile and context.

```
@ContextConfiguration(classes = {TestContainerPGConfig.class})
@ActiveProfiles({TestContainerPGConfig.TESTCONTAINER_PG})
public class DBBaseIT extends BaseIT {

}
```

Even for this simple test scenario, execution takes much less time.



This was one simple showcase how we could easily integrate Unit tests and DockerTestCotnainers. Since there are many modules of testcontainers out there, we could easily find right for many applications and systems we integrate in our systems. Test containers makes life easier and test application like a charm.