Quarkus Tests mit TestContainers?:

Was ist überhaupt ein TestContainer?:

Testcontainers ist eine Java-Bibliothek, die JUnit-Tests unterstützt und einfache, wegwerfbare Instanzen gängiger Datenbanken, Selenium-Webbrowser oder alles andere bereitstellt, das in einem Docker-Container ausgeführt werden kann.

Testcontainer erleichtern die folgenden Arten von Tests:

- **Integrationstest für die Datenzugriffsschicht (Beispiel):**
  Was ist Integrationstest(Theorie):
  Integrationstests überprüfen die Zusammenarbeit mehrerer Systemteile zunehmender Komplexität von einzelnen Modulen über Teilsysteme bis zum Gesamtsystem.
  Dabei muss sowohl die korrekte Interaktion, wie z.B. der Austausch von Daten durch Nachrichten oder gemeinsam benutzten Speicher, der Zugriff auf Datenbanken oder die Nutzung von Funktionalität durch Aufrufe von Schnittstellenfunktionen über einzelne Teile hinaus, überprüft werden als auch das Nicht-Auftreten unerwünschter Effekte. Integrationstests werden wegen des vornehmlich System-internen Fokus beim Ersteller der Software durchgeführt.

  Was bedeutet genau Intgrationstest für die Datenzugriffsschicht: Man soll eine containerisierte Instanz einer MySQL-, PostgreSQL- oder Oracle-Datenbank verwenden, um unseren Code für die Datenzugriffsschicht auf vollständige Kompatibilität zu testen, ohne dass eine komplexe Einrichtung auf den Maschinen der Entwickler erforderlich ist,
  und mit der Gewissheit, dass Ihre Tests immer mit einem bekannten DB-Zustand beginnen werden. Jeder andere Datenbanktyp, der containerisiert werden kann, kann ebenfalls verwendet werden.

- **Application Integration Tests(Anwendungsintegrationstests):**
  für die Ausführung der
  Application in einem kurzlebigen Testmodus mit Abhängigkeiten wie Datenbanken, Nachrichtenwarteschlangen oder Web-Servern.

- **UI/Acceptance Tests:**
  Akzeptanztest:
  Ein Akzeptanztest oder Abnahmetest) ist in der Softwaretechnik die Überprüfung, ob eine Software aus Sicht des Benutzers/Kunden wie beabsichtigt funktioniert und dieser die Software akzeptiert.

  Verwendung containerisierter Web-Browser, die mit Selenium kompatibel sind, zur Durchführung automatisierter UI-Tests. Jeder Test kann eine frische Instanz des Browsers erhalten, ohne dass man sich um den Browserstatus, Plugin-Varianten oder automatische Browser-Upgrades kümmern muss. Und Sie erhalten eine Videoaufzeichnung von jeder Testsitzung, oder einfach von jeder Sitzung, in der die Tests fehlgeschlagen sind.

Modultest: Testgegenstand ist die Funktionalität innerhalb einzelner abgrenzbarer Teile der Software (Module, Programme oder Unterprogramme, Units oder Klassen).

Integrationstest: Der Integrationstest bzw. Interaktionstests testet die Zusammenarbeit voneinander abhängiger Komponenten. Der Testschwerpunkt liegt auf den Schnittstellen der beteiligten Komponenten und soll korrekte Ergebnisse über komplette Abläufe hinweg nachweisen.

Systemtest: Der Systemtest ist die Teststufe, bei der das gesamte System gegen die gesamten Anforderungen (funktionale und nicht funktionale Anforderungen) getestet wird. Gewöhnlich findet der Test auf einer Testumgebung statt und wird mit Testdaten durchgeführt. Die Testumgebung soll die Produktivumgebung des Kunden simulieren. In der Regel wird der Systemtest durch die realisierende Organisation durchgeführt
Abnahme/Akzeptanztest: Ein Abnahmetest, Akzeptanztest oder auch User Acceptance Test (UAT) ist ein Test der gelieferten Software durch den Kunden bzw. Auftraggeber. Oft sind Akzeptanztests Voraussetzung für die Rechnungsstellung. Dieser Test kann bereits auf der Produktivumgebung mit Kopien aus Echtdaten durchgeführt werden

Rest-Assured

Mit dem Framework Rest-Assured lassen sich REST-Schnittstellen zuverlässigen testen.

Damit REST Assured bei den Tests zur Verfügung steht, muss folgende dependecy zur pom.xml hinzugefügt werden.

Prinzipiell sind REST Assured Tests immer nach dem gleichen *AAA*-Muster aufgebaut.

Mit given() wird ein Testszenario beschrieben (*ARRANGE(anordnen)*). Mit when() wird beschrieben, welche Operationen ausgeführt werden soll (*ACT(Handlung)*). Und mit then() wird überprüft, ob das Ergebnis den Annahmen entspricht (*ASSERT(geltend machen)*).

Beispiel Quarkus Tests mit TestContainers:

1. Pom.xml herzeigen

```xml
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>1.12.5</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>postgresql</artifactId>
  <version>1.12.4</version>
  <scope>test</scope>
</dependency>
```

Um testcontainers nutzen zu können braucht man mindestens die dependency junit-jupiter, jedoch verwenden wir in diesem Beispiel eine PostgreSQL-Container verwenden und damit sind wir nicht die einzige, deshalb stellt Testcontainers für gängige Container bereits fertige APIs zur Verfügung

Somit können wir anstelle des GenericContainer direkt die
Rule PostgreSQLContainer verwenden

2. Programm vorstellen
   Nun zum Code. Ich habe die Entity Developer die Panache extended(erweitert).
   (Unique = Es wird kein Wert doppelt abgespeichert.)

   Dazu habe ich ein DAO erstellt das PanacheRepository implementiert.
   @Singleton: Es stellt sicher, dass von einer Klasse genau ein Objekt existiert
   Dank Panache sind (fast) alle JPQL-Queries(Abfragen von Entitäten) überflüssig. Einfache Abfragen können ganz leicht mithilfe des Repositories gemacht werden. Dadurch verliert man eine riesige Fehlerquelle.

   (Instanz: mit dependency injection wird eine neue Instanz angelegt)

   Weites habe ich noch eine Klasse DeveloperEndpoints erstellt.
   Dazu habe ich DeveloperDao injected(einspeisen) und 3 Methoden erstellt, findDeveloper @Get, createDeveloper @Post und deleteDevloper @Delete

   Application.properties:
   Configuration für postgres, die man später noch brauchn werden

   Um jetzt diese http Methoden zu testen habe ich mir eine Klasse DeveloperResourceTest erstellt vorher zeige ich Ihnen jedoch die Klasse TestContainersPostrgesqlTestResource.

   Diese Klasse implementiert QuarkusTestResourceLifecycleManager. Dieses Interface erlaubt uns zu definieren was man vor allen Tests ausführen soll und was nach allen Tests ausgeführt werden soll. Dabei mein ich nicht nur Testmethoden sondern auch Test klassen.

   Dann erstellt man das field PostgresqlContainer, mit der Methode configurationParameters wird eine Map returned mit den configurationen der PostgreSQL Datenbank.
   Start wird überschrieben und gibt eine Map zurück mit den properties die ich nutzen will.

   Man sieht schon das ich eine innere statische klasse habe und eine public class TestContainersPostgresSqlTestResource.
   Mit der Annotation @QuarkusTestResource, mit der sage ich dem Test runner bevor er die Tests aufrufen soll, soll er die Start methode aufrufen und nach allen Test die Stop methode

   Compilieren: mvnw compile test

   Checking the System …. Diese Ausgabe ist nicht von Quarkus sondern von Testcontainers.
   (Die Docker-Umgebung sollte mehr als 2 GB freien Speicherplatz haben.)
   Mit sieht auch localhost hat nicht den port 5432 sondern einen anderen der immer zufällig ist. Um auch andere Testcontainers auf der selben machine laufen lassen zu können vergibt Testcontainers die Ports zufällig. Wenn man den port ändert kann es binding problems geben.

Fazit:

Viele werden sich wahrscheinlich fragen warum man nicht einfach die Datenbank mockt. Mocking ist hier oft keine Option, da man das tatsächliche System testen möchte. Die Tests laufen anschließend gegen eine wirklich richtige Instanz der gewählten Datenbank und erhöhen somit das Vertrauen, dass der so getestete Code auch wirklich funktioniert.

Abseits von Datenbanken lassen sich Testcontainers auch für Tests, die einen laufenden Browser benötigen, einsetzen. Vor allem der Aufwand, den passenden Browser lokal zu installieren, entfällt hierbei und bietet somit einen klaren Vorteil.

Blackbox-Systemtests mit injiziertem MicroProfile REST-Client funktioniert

Unterschied WhiteBox Testing und BlackBox Testing:

WhiteBox Testing:
Hier haben die Tester Zugriff auf und Kenntnis über die Entwicklung der Software (Quellcode, Entwicklungsumgebung, Diagramme [UML etc.] sowie Dokumentation). Da quasi in das Programm "hineingesehen" wird, wird auch seltener vom "Glass-Box-Test" gesprochen.

BlackBox Testing:
Hier haben die Tester keinen Zugriff auf und idealerweise auch keine Kenntnis über die Entwicklung der Software.

1. Stufe: Tester, welche als Kunden auftreten (echter Kunde oder nur "simulierter" Kunde) testet die Software (vor allem Funktionstests)
2. Stufe: Tester, welche Erfahrung mit Software-Tests (z. B. Penetration Tests) haben und sich in die Auftragsspezifikationen hineinarbeiten, führen den Test durch (vor allem Stresstests)

Nun zum Beispiel ich hab einen microservice erstellt mit einer einfachen rest ressource mit den path hello, und ein weiteres Projekt zum testen(hello-st für systemtests)

Zu der pom rest-client hinzufügen

Dann habe ich ein Interface erstellt HelloRessource damit registriere ich meine RestClient und definiere meine einzige Methode String content().

Danke für ihre Aufmerksamkeit!

Quellen:

https://www.testcontainers.org/
https://www.youtube.com/watch?v=xtQ1GmVpPP0
https://www.youtube.com/watch?v=RLkvFT537_4&t=14s
https://github.com/quarkusio/quarkus/blob/master/test-framework/common/src/main/java/io/quarkus/test/common/QuarkusTestResourceLifecycleManager.java
https://www.hameister.org/SpringMvcRESTassured.html
https://in.relation.to/2019/11/19/hibernate-orm-with-panache-in-quarkus/
https://joel-costigliola.github.io/assertj/assertj-core-quick-start.html