Erkki Keränen

REST-rajapinnan toteutus ja yksikkötestaus Java EE -teknologialla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

1.2.2017

Metropolia

Tekijä Otsikko Sivumäärä Aika	Erkki Keränen REST-rajapinnan toteutus ja yksikkötestaus Java EE -teknologialla 50 sivua + 12 liitettä 1.2.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaajat	Ohjelmistosuunnittelija Seppo Pääjärvi Yliopettaja Jarkko Vuori

Insinöörityössä toteutettiin REST-rajapinta vakuutusrekisterijärjestelmään Java EE -teknologialla. Toteutukselle kirjoitettiin yksikkötestit, joiden tarkoitus oli varmistaa kirjoitetun ohjelman laatua ja toimintavarmuutta sekä auttaa tulevaisuuden muutoksien heijastusvaikutusten selvittämisessä.

Jokaiselle luokalle tehtiin omat yksikkötestit omiin yksikkötestiluokkiin. Iteratiivisen kehitysprosessin päätteeksi sovelluksen arkkitehtuuria parannettiin yksinkertaistamalla, jolloin yksikkötestien avulla voitiin todeta, että sovellus edelleen toimitti tehtävänsä.

Kehitysympäristönä käytettiin Eclipse Mars -sovellusta. Ennestään olemassa oleva Vakuutusrekisteri-sovellus ja siihen tässä insinöörityössä toteutettu REST-rajapinta toteutettiin Javalla. REST-rajapinta toteutettiin JAX-RS RESTEasy -implementaatiolla. Yksikkötestit toteutettiin JUnit-kirjastolla, ja yksikkötestien riippuvuuksia sisältävien kenttien oliot injektoitiin Mockito-kirjaston mock-olioilla.

Sovellettu toteutustapa yksikkötesteineen palveli tarkoitusta, jolla insinöörityön konkreettisen työvaiheen suoritus saatiin toteutettua varmasti ja tulevaisuuden muutoksia kestäväksi.

Avainsanat	REST, JAX-RS, RESTEasy, Java, Java EE, JUnit, mock, Mockito

Author	Erkki Keränen
Title	REST API development and unit testing in Java EE
Number of Pages	50 pages + 12 appendices
Date	1 February 2017
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Software Engineering
Instructors	Seppo Pääjärvi, Software Designer Jarkko Vuori, Principal Lecturer

The goal of this final year project was to develop a REST API with Java EE technology for an insurance database system. To guarantee quality, stability and maintenance ability, unit tests were written for the feature developed during the implementation.

All classes introduced as part of the new feature were unit tested. The architecture of the application was streamlined towards the end of the iterative development process. Refactoring of the application because of streamlining was confirmed not to have side effects with the help of the written unit tests.

Eclipse Mars IDE was used for writing the software and running the unit tests. The already existing application Vakuutusrekisteri and the developed REST API was written in Java and the REST API was developed using the JAX-RS RESTEasy implementation. Unit tests were made and run using JUnit. Integration dependencies contained in the new code were tested by utilizing the Mockito mocking framework mock objects.

The result was a completed and deliverable feature including complete unit tests providing protection against regression.

The chosen unit testing approach was appropriate, providing quality control over produced work during development and providing protection against possible errors caused by future modifications.

Keywords	REST, JAX-RS, RESTEasy, Java, Java EE, JUnit, mock, Mockito

Författare Arbetets namn Sidantal Datum	Erkki Keränen Utveckling och enhetstestning av REST-gränssnitt med Java EE -teknologi 50 sidor + 12 bilagor 1.2.2017
Examen	Ingenjör (YH)
Utbildningsprogram	Informationsteknik
Inriktning	Datateknik
Handledare	Systemutvecklare Seppo Pääjärvi Överlärare Jarkko Vuori

I detta examensarbete utvecklades ett REST-gränssnitt för ett försäkringsregister med hjälp av Java EE -teknologi. Under förverkligandet av programmet skrevs enhetstest, vars uppgift var att försäkra programmets kvalitet, tillförlitlighet och möjlighet för förändringar i den eventuella framtiden.

För varje klass skrevs enhetstest i separata enhetstestklasser. Genom den hela iterativa utvecklingsprocessen kunde programmets ändamålsenliga funktion verifieras, även om programvarans arkitektur ändrades och förenklades på vägen mot den slutliga versionen.

Som integrerad utvecklingsmiljö (IDE) användes Eclipse Mars. REST-gränssnittet som utvecklades i detta examensarbete för den redan existerande Vakuutusrekisteri-programmet förverkligades med Java. För implementeringen av REST-gränssnittet valdes JAX-RS RESTEasy-implementationen. Enhetstesten utvecklades med hjälp av JUnit-ramverket. Mockito-ramverket användes till att injektera mock-objekt för enhetstest. Mock-objekten spelade en betydande roll i enhetstestningen av klasser som hade externa besittningar.

Valet över framställningssättet var lyckat. Programmeringsarbetet som utfördes kunde konstateras vara enligt uppdragsgivarens krav. Med stöd av de skrivna enhetstesten, ansågs det realiserade programmet tåla möjliga förändringar i framtiden.

Nyckelord	REST, JAX-RS, RESTEasy, Java, Java EE, JUnit, mock, Mockito

Sisällys

Liitteet

Liite 1. src/main/java/model/Entities.java

Liite 2. src/main/java/model/Entity.java

Liite 3. README.md – Esimerkkiprojektin kuvaus ja ohjeet

Liite 4. pom.xml – Esimerkkiprojektin projektimalli

Liite 5. src/main/java/rest/JaxRsActivator.java

Liite 6. src/main/java/rest/RestService.java

Liite 7. src/main/java/data/DataRepository.java

Liite 8. src/test/java/engineering/DataRepositoryJUnitTest.java

Liite 9. src/test/resources/arquillian.xml

Liite 10. src/test/java/engineering/DataRepositoryArquillianTest.java

Liite 11. src/test/java/engineering/RestServiceArquillianTest.java

Liite 12. src/test/java/engineering/RestServiceJUnitTest.java

Lyhenteet

API	*Application Programming Interface.* Ohjelmointirajapinta, jonka avulla ohjelma tarjoaa muille ohjelmille ominaisuuksien käyttömahdollisuuden.
Black box	Ohjelmistotestaustapa, jossa testattavan osan sisäinen rakenne on testaajalle tuntematon.
CRUD	*Create, Read, Update and Delete.* Tiedon pysyvän tallentamisen neljä perusfunktiota.
DTO	*Data Transfer Object.* Tiedonmuunto-olio, jota käytetään tiedonmuuntamisessa olioluokasta toiseen.
Gray box	Ohjelmistotestaustapa, jossa testattavan osan sisäinen rakenne on tunnettu, mutta testaus tehdään Black box -periaatteella.
HATEOAS	*Hypermedia As The Engine Of Application State.* REST-määritelmä, joka määrittelee, että asiakas kuluttaa palvelimen palveluja suoraan hypermedian kautta.
HTML	*HyperText Markup Language.* Merkintäkieli, jolla kuvataan hypermediadokumentteja.
HTTP	*HyperText Transfer Protocol.* Siirtoprotokolla, jolla siirretään hypertekstiä (esim. internetsivut)
Hypermedia	Epälineaarinen dokumentti, joka voi myös sisältää upotettua kuvaa ja ääntä tekstin lisäksi sekä linkkejä muihin sivuihin. Esimerkiksi HTML-dokumentti.
JSON	*Javascript Object Notation.* Tekstipohjainen tiedon esitysmuoto, jossa tieto esitetään attribuutti-arvopareina.
REST	*Representational State Transfer.* Internetin HTTP-protokollalla toteutettujen palvelujen yleinen API-arkkitehtuurimalli.

RESTful	Palvelu, joka toteuttaa REST-arkkitehtuurin mallin.
SUT	*System Under Test*. Yksikkötestissä testattavana oleva kokonaisuus, esimerkiksi funktio tai metodi.
TDD	*Test Driven Development.* Ohjelmistokehitystapa, jossa testaus toimii ohjelmoinnin ajurina.
URI	*Uniform Resource Identifier.* Merkkijono, joka tunnistaa tietyn resurssin. Voi olla relatiivinen tai sisältää tiedon esitysmuodosta ja sijainnista.
URL	*Uniform Resource Location.* Merkkijono, jonka perusteella voidaan tunnistaa tietty resurssi, esitysmuoto ja sijainti verkossa.
URN	*Uniform Resource Name.* Merkkijono, joka tunnistaa tietyn resurssin. Informaatio on hyödytön, ellei tiedä, missä kontekstissa tietoa käytetään.
White box	Ohjelmistotestatustapa, jossa testattavan osan sisäinen rakenne tunnetaan.
WWW	World Wide Web. Palvelu, jolla julkaistaan ja hyödynnetään dokumentteja verkossa.
XML	*Extensible Markup Language.* Tekstipohjainen tiedon esitysmuoto, rakenteellinen kuvauskieli, jonka skeeman avulla voidaan kuvata itse tieto ja sen muoto.

1 Johdanto

Kun kehitetään ohjelmia, jotka keskustelevat autonomisesti tietoverkon välityksellä keskenään, tarvitaan käyttökelpoista ohjelmointirajapinta-arkkitehtuuria. Käyttökelpoisella tässä tapauksessa tarkoitetaan ohjelmointiystävällistä, jolloin ohjelmistokehittäjä voi valjastaa toisen ohjelman tarjoamat funktiot omaan käyttöönsä. Tämänkaltaisessa ja monessa muussa tapauksessa on kaikkien edun mukaista, että rajapinta toteutetaan joko virallisen standardin tai de facto -standardin mukaisesti.

Jos ei ainoastaan pohdita työn mielekkyyden ja tuottavuuden kannalta, tulee ottaa huomioon paljon enemmän tekijöitä kuin pelkästään pyynnön toteutuminen annetun käskyn mukaisesti kertaluontoisena kehitystyönä. Kun kysymyksessä on palvelu, jota eri tietojärjestelmät käyttävät eri asiakasnäkökulmista verkon välityksellä, tulee toteutuksessa ottaa huomioon samaa tietoa sisältävän vastauksen soveltuvuus eri asiakasjärjelmien kyselyihin. Jos järjestelmään toteutetaan asiakaskohtaiset ohjelmointirajapinnat, tulee arkkitehtuuri toteuttaa niin, että se on mahdollisimman samankaltaisesti hyödynnettävissä kaikilla osapuolilla. Koska eri asiakasjärjestelmillä on erilaisia vaatimuksia vastauksen esitysmuodon tai yksityiskohtaisuuden kannalta, on hyvä jollain tavalla varmistaa eri versioita toteuttaessa, että muutokset eivät vaikuta tiedon oikeellisuuteen tai esitysmuotoon.

Elisa Appelsiini on toteuttanut käytössä olevan Vakuutusrekisteri-järjestelmän Tapaturmavakuutuskeskukselle. Tapaturmavakuutuskeskus ylläpitää vakuuttamisen valvontaa, harmaan talouden torjuntaa ja korvausvastuullisen vakuutusyhtiön selvittämistä varten rekisteriä työnantajista, joilla on tai on ollut pakollisia vakuutuksia työntekijöitään varten. Vakuutusrekisteriin on tarkoitus toteuttaa useita rajapintoja eri viranomaisille ja tahoille, ja tämän insinöörityön tehtävänä on toteuttaa HTSY:lle (Harmaan talouden selvitysyksikkö) REST-rajapinta Vakuutusrekisteri-järjestelmään. Tämän rajapinnan avulla HTSY:n järjestelmästä voidaan luoda kyselyjä, jotka palauttavat tiedon pyydettyjen yritysten voimassa olevista tapaturmavakuutuksista.

Elisa Appelsiini on Elisan IT-liiketoimintayksikkö, jonka palveluihin kuuluu muun muassa tietotekniikka- ja sovelluskehitys. Liiketoimintayksikkö työllisti n. 350 henkeä vuonna 2014. (1.)

Tässä insinöörityössä toteutetaan Java EE -teknologialla REST-ohjelmointirajapinta ja toteutuksen yksikkötestaus osana kehityksen laadunvarmistusta. REST on lyhenne sanoista Representational State Transfer, joka on WWW-sovellusarkkitehtuuria ohjaava arkkitehtuuri (2, s. 107). Aluksi raportissa käydään läpi REST-arkkitehtuuriin liittyvät peruskäsitteet, ja sitten pureudutaan rajapinnan toteuttamiseen Java EE:llä ja syvennytään toteutuksen yksikkötestaukseen.

Insinöörityön toteutuksen tulosta arvioidaan niiden huomioiden pohjalta, joita kirjattiin muistiin toteutuksen edetessä.

Insinöörityössä esitellään myös käsitteiden englanninkieliset termit, koska on todennäköistä, että tällä ongelma-alueella työskentelevä henkilö on tekemisissä niiden kanssa alan kirjallisuudessa ja kehitystyökaluissa.

2 REST-ohjelmointirajapinta

API on lyhenne sanoista *Application Programming Interface*, joka suomennettuna tarkoittaa ohjelmointirajapintaa. REST-arkkitehtuuri on kehitetty ohjaamaan WWW-arkkitehtuurin kehitystä (2, s. 105–106). Koska WWW-arkkitehtuuri itsessään nojaa pitkälle REST-arkkitehtuuriin, REST-rajapinnan käyttäminen ei itsessään välttämättä tarvitse ohjelmointia sanan varsinaisessa merkityksessä (vrt. ohjelmakoodin kirjoittamista ja kääntämistä), vaan mitä tahansa rajapintaa (verkkosivua) voidaan hyödyntää suoraan esimerkiksi selaimella tai muulla soveltuvalla asiakasohjelmalla, kunhan verkkosivu noudattaa REST-periaatteita (3, luku 2, osio 1).

Ohjelmointirajapinta on joukko dokumentoituja tai standardoituja menetelmiä muille ohjelmille kuluttaa ohjelman palveluita. REST-ohjelmointirajapinta on täten jonkin sovelluksen HTTP-kutsuihin vastaava rajapinta, joka suorittaa jotain, mikä vastaa kutsun sisältöä. HTTP on lyhenne sanoista HyperText Transfer Protocol eli hypertekstin siirtoprotokolla. Sen avulla WWW-sovellukset siirtävät ja päivittävät tietoa.

REST-ohjelmointirajapinta voi tarjota asiakkaalle mahdollisuuden tiedon käsittelemiseen, hakemiseen tai tallentamiseen. Lisäksi rajapinnan kautta voi toteuttaa esimerkiksi järjestelmän ohjauksen ja tilan kyselyn.

2.1 REST-arkkitehtuurin määritelmä

REST on arkkitehtuurimääritelmä hajautetulle hypermediajärjestelmälle, joka korostaa muun muassa skaalautuvuutta, rajapintojen esitystavan yleistämistä, komponenttien itsenäisyyttä ja tietoturvan toimeenpanoa (2, s. 105). REST ei ole standardi, protokolla tai tiedostomuoto, eikä sille ole olemassa tiettyä ohjelmistokehityskirjastoa. REST on kokoelma määritelmiä, jotka kuvaavat muun muassa optimaalisen WWW-palvelun toteutuksen, joka pitää sisällään tilattomuuden (Stateless), HATEOAS:n (Hypermedia as the Engine of Application State) ja muita suunnitelmallisia määritteitä (3, luku 3, kappale 2).

Tilattomuudella tarkoitetaan sitä, että palvelin tai palvelua tuottava järjestelmä ei tiedosta eikä sille ole merkitystä, mikä on kyselyn esittävän asiakkaan tila ja päinvastoin. Kysyjä tai vastaaja ei tee olettamuksia tai päätöksiä sen perusteella, missä tilassa toinen mahdollisesti on, vaan periaate on se, että kyselyn voi aina tehdä, minkä jälkeen siihen voi aina vastata. (3, luku 1, osio Short Sessions.)

HATEOAS taas kuvaa sitä, että esitetty hyperteksti, juuri nyt esillä oleva verkkosivu, esimerkiksi verkkopankin etusivu, kertoo asiakkaalle, mihin tilaan hän voi siirtyä seuraavaksi, esimerkiksi laskun maksamiseen (3, osio Connectedness, kappale 3).

REST-ohjelmointirajapintaa ei yleensä ole tarkoitus käyttää suoraan ihmisasiakkaan toimesta internet-selaimella. Voidaan monesti nähdä, että oikein muotoiltu kysely REST-ohjelmointirajapintaan palauttaa kyselylle kuuluvan sisällön, mutta ei sisällä esimerkiksi navigointilinkkejä tai tietoa siitä, mitä asiakas voisi seuraavaksi tehdä. HATEOAS-periaatetta voidaan noudattaa vastauksissa, mutta vastauksessa palautuvat lisätiedot olisivat tapauskohtaisia ja esimerkiksi kyselyä tekevän sovelluksen tulisi itse kyetä päättelemään, mitä tiedolla tehtäisiin.

HTTP ja REST

Koska REST-rajapintaa käytetään HTTP:llä, kuten aikaisemmin mainittiin, on hyvä huomioida, että periaatteessa monikin REST-ohjelmointirajapinta on käytettävissä suoraan internet-selaimella, vaikka niitä ei yleensä sellaisenaan ihmisasiakkaille ole käytettäväksi tarkoitettu. Selaimella ei välttämättä ole tarjota käyttäjälle suoraan mahdollisuutta vapaasti muodostaa kyselyjen parametreja. REST-rajapintaa hyödyntävät sovellukset ovat yhteydessä HTTP:llä rajapintaan ja käyvät HTTP-standardin mukaista vuoropuhelua.

URL ja URI

Koska REST-kutsun oleellisena osana on sijainti, käydään tässä lyhyesti läpi, millä terminologialla osoiterivin elementtejä tai osia käsitellään.

Uniform Resource Locator eli URL on skeema, jolla kuvataan tietoverkossa olevan resurssin tarkka osoite (4, s. 19).

Uniform Resource Identifier eli URI voi myös kuvata tarkasti tietoverkossa olevan resurssin yksilöllisen osoitteen, mutta sen sallitaan myös olevan relatiivinen tai merkitsevän tiettyä asiaa tietyssä asiayhteydessä (5, s. 1 ja 8).

HTTP:llä tavoitettavan resurssin (URL tai URI) osoite koostuu esimerkiksi seuraavanlaisesti:

http://[palvelimen osoite][: TCP-portti]/[polku[?avain1=arvo1&avain2=arvo2]]
(5, s. 16; 4, s.19; 6, s. 9 ja 3)

esimerkkinä URL : http://www.palvelin.com:80/palvelut?jarjestys=hinta

esimerkkinä URI : http://www.palvelin.com:80/palvelut/1 tai palvelut?jarjestys=hinta

CRUD

Usein palvelut, jotka tarjoavat REST-ohjelmointirajapintaa, tarjoavat mahdollisuuden luoda, lukea, päivittää ja poistaa tietoa jostain tietovarastosta. HTTP:n POST-, GET-, PUT- ja DELETE-operaatiot toteuttavat edellä mainittuja toimintoja. Tätä paradigmaa kutsutaan CRUD-paradigmaksi. Tietovarastona toimii yleensä tietokanta, joka tukee CRUD-paradigman mukaisia toimintoja (Create, Read, Update and Delete). (7, s. 381; 8.)

Periaatteessa REST-ohjelmointirajapinnasta voidaan suoraan esittää tietokannan tietomallia sellaisenaan ja tarjota CRUD-operaatioita, mutta se ei ole suotavaa. On suositeltavaa, että REST-ohjelmointirajapinta, jolla on asiakkaita, kykenee toimittamaan CRUD-operaatiot riippumatta esimerkiksi siitä, muuttuuko tallennusjärjestelmässä oleva tietomalli. (3, luku 9, osio Don’t Fall into the Collection Trap.)

2.2 JSON- ja XML-formaatit

Kun REST-ohjelmointirajapinta mahdollistaa CRUD-operaatiot, on käytettävä sellaista tiedon esitysmuotoa, jota molemmat osapuolet ymmärtävät. Esimerkkejä esitysmuodoista ovat HTML (HyperText Markup Language), JSON (JavaScript Object Notation), XML (Extensible Markup Language) tai jokin tiettyä sovellusta varten räätälöity tiedonesitysmuoto. Tavoitteena on yleensä, että vastauksessa palautettu tieto voidaan automaattisesti käsitellä ohjelman jossain komponentissa esimerkiksi esittämistä tai analysointia varten. Tästä syystä Javassa ja vastaavissa ohjelmointiteknologioissa on kirjastoja olioiden ja tietorakenteiden muuntamiseen XML- ja JSON-muotoihin sekä päinvastoin.

Seuraavaksi esiteltävissä koodiesimerkeissä 1 ja 2 kuvataan kaksi Java-luokkaa, joiden JSON- ja XML-muunnokset esitetään koodiesimerkeissä 3 ja 4.

```
@XmlRootElement (name = "Entities")
public class Entities {
  @XmlElement(name = "Entity")
  public List<Entity> entities;
  public Entities() {
    entities = new ArrayList<Entity>();
  }
}
```

Koodiesimerkki 1. Entities-luokka joka sisältää taulukon Entity-olioita. Ohjelmakoodi on kokonaisuudessaan liitteessä 1.

```
@XmlRootElement(name = "Entity")
public class Entity {
  private int id;
  private String name;

  @XmlElement
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  @XmlElement
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

Koodiesimerkki 2. Entity-luokka. Ohjelmakoodi on kokonaisuudessaan liitteessä 2.

Koodiesimerkissä 1 ja 2 kuvattiin Entities-luokka, joka sisältää taulukon Entity-olioita. Näitä luokkia käytetään vastaavan JSON- ja XML-muodon esityksen havainnollistamisessa koodiesimerkeissä 3 ja 4. Java-annotaation @XmlRootElement avulla määritellään itse luokat palautettavaksi, ja annotaation @XmlElement avulla voidaan määritellä, onko itse kenttä tai sen getter-metodi se, mikä vastauksessa palautetaan hierarkkisesti juurielementin alla.

Seuraavaksi käsitellään lyhyesti JSON ja XML sekä havainnollistetaan niiden esitysmuodot taulukossa 1 kuvatun tiedon arvoilla. Taulukko on Entities-luokan Entities-taulukko, ja taulukon rivien arvosarakkeet vastaavat yhtä Entity-oliota. Esimerkkeinä annettavat JSON- ja XML-vastaukset luotiin myöhemmin esiteltävien koodiesimerkkien 6–8 toteuttamalla sovelluksella.

Taulukko 1. Entities-taulukon sisältö koodiesimerkkejä 3 ja 4 varten.

Taulukon alkio	id-kentän arvo	name-kentän arvo
1	0	Matti Meikäläinen
2	1	Tanja Teikäläinen

JSON voidaan nähdä kevyenä ja suoraviivaisena tapana esittää tietoa ja tietorakenteita. JSON-dokumentissa voidaan esittää primitiivisiä tietotyyppejä, kuten boolean, merkkijono, luku, null, sekä strukturoituja tietotyyppejä olio ja taulukko. Tieto esitetään avain-arvo-pareina. (9, s. 1.)

Koodiesimerkissä 3 annetaan esimerkki koodiesimerkkien 1 ja 2 Java-luokista taulukon 1 tiedoilla JSON-muodossa.

```
{
  "Entity": [
    {
      "id": 0,
      "name": "Matti Meikäläinen"
    },
    {
      "id": 1,
      "name": "Tanja Teikäläinen"
    }
  ]
}
```

Koodiesimerkki 3. Entity-taulukon (Entitities-olio) palautus JSON-muodossa.

XML on tiedosta riippuen JSON:a raskaampi rakenteeltaan, mutta iältään kypsempi teknologia. XML:ssä on standardoitu mahdollisuus skeeman määrittelyyn. XML Schema 1.0

julkaistiin ja standardoitiin W3C:n (World Wide Web Consortium) toimesta vuonna 2001. (10)

XML-skeemadokumentti on määrämuotoinen tapa kuvata tiedon rakenne käyttäen ennalta määriteltyä sanastoa ja XML-muotoa (10). Voidaan sanoa, että skeema on malli, jonka avulla jokin asia esitetään tietyllä tavalla.

XML-dokumentin tapauksessa XML-skeeman avulla voitaisiin ohjelmallisesti muotoilla ja todeta, että lähetetty tai vastaanotettu dokumentti on muotoiltu ja täytetty oikein (11, s. 1). Koodiesimerkissä 4 on esitetty Entities-olio XML-muodossa ja koodiesimerkissä 5 Entities-olion XML-skeema.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Entities>
  <Entity>
    <id>0</id>
    <name>Matti Meikäläinen</name>
  </Entity>
  <Entity>
    <id>1</id>
    <name>Tanja Teikäläinen</name>
  </Entity>
</Entities>
```

Koodiesimerkki 4. Entities-olion palautus XML-muodossa.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Entities" type="entities"/>
  <xs:element name="Entity" type="entity"/>
  <xs:complexType name="entities">
    <xs:sequence>
      <xs:element ref="Entity" minOccurs="0" maxOccurs="un-
bounded"/>
    </xs:sequence>
```

```
  </xs:complexType>
  <xs:complexType name="entity">
    <xs:sequence>
      <xs:element name="id" type="xs:int"/>
      <xs:element name="name" type="xs:string" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
</xs:schema>
```

Koodiesimerkki 5. Schemagen-sovelluksella luotu XML-skeema Entity- sekä Entities-luokista.

Koodiesimerkissä 4 on kuvattu koodiesimerkkien 1 ja 2 oliot esitettynä XML-muodossa. Koodiesimerkkejä 3 ja 4 vertaillessa voi huomata, että rakenteellisen tiedon esitys JSON-muodossa vie vähemmän tilaa, mikä voi olla merkittävä asia, kun tietoa siirretään tietoverkossa. Pientä optimointia olisi voitu tehdä XML:n suhteen esittämällä id-kenttä olion attribuuttina, mutta koodiesimerkkien vertailun helpottamiseksi tätä ei tehdä. Toisaalta XML:n etuna on yleisesti käytössä oleva standardi skeemalle, jonka avulla osapuolet voivat tarkastaa dokumentin rakenteen oikeellisuuden.

JSON:a varten on myös kehitetty skeema, jonka uusin julkaisu löytyy internetistä osoitteesta www.json-schema.org. JSON-skeema mahdollistaa tiedon rakenteen kuvauksen sekä muun muassa tietotyypit ja vähimmäisarvot. IETF:n (Internet Engineering Task Force) standardointiprosessi on tämän insinöörityön kirjoitusajankohtana kesken ja dokumentoinnin vedos vanhentunut vuonna 2013. (12, s. 1–2.)

Voidaan kuitenkin olettaa, että REST-ohjelmointirajapintaa hyödyntävät asiakkaat voivat toteuttaa kutsunsa millä tahansa teknologialla ja on edullista pysytellä yleisesti käytössä olevissa dokumenttimuodoissa, joille löytyy laaja tuki ohjelmointikielistä tai niiden kirjastoista. Tällöin JSON, XML tai molemmat ovat luonnollisia valintoja. REST-rajapinta voi tarjota tekijän soveltamaa sisältömuotoa tai mahdollisuuden asiakkaan pyytää tiettyä esitysmuotoa.

Joka tapauksessa REST-rajapinnan HTTP-vastauksessa Content-Type-kentässä tulee kertoa asiakkaalle sisällön esitysmuoto, esimerkiksi application/json tai application/xml (4, s. 124). Vastaavasti asiakas voi pyytää haluamansa vastausmuodon Accept-kentässä (4, s. 100).

2.3 Java EE ja REST-arkkitehtuurimalli

REST-ohjelmointirajapinnat ovat yleistyneet merkittävästi, jopa niin, että suuret palveluntarjoajat, kuten esimerkiksi Google, Twitter ja Facebook, tarjoavat kyseisenlaista ohjelmointirajapintaa palveluidensa käyttöön. Vastaavasti useat ohjelmointikielet ja teknologiat kuten esimerkiksi Java, .NET, Node.js, Ruby, Python, PHP ja JavaScript antavat mahdollisuuden hyödyntää REST-rajapintaa sekä osa kykenee tarjoamaan sitä palvelinsovelluksena. Tämä voidaan nähdä suoraviivaisena ja yleisesti saatavilla olevana toteutustapana, koska REST-rajapinnoissa on tarjontaa, tukea ohjelmointikielissä ja internetin myötä laaja-alaisesti vakiintunut HTTP-protokolla.

Yhtenä syynä REST-arkkitehtuurin soveltamisen suureen suosioon nyt ja tulevaisuudessa lienee se, että REST-määritelmä itsessään pitää sisällään yhden onnistuneimmista ihmisen keksimistä teknologioista, World Wide Webin, ja perustuu siihen (3, luku 3, kappale 3).

Koska insinöörityön toteutus tehtiin Java-teknologialla, esitellään tässä raportissa Java-kielen ohjelmointirajapinta REST-rajapintojen kehittämiseen. JCP:n (Java Community Process) tuottama JSR-000339-spesifikaatio (Java Specification Request) kuvaa JAX-RS: Java API for RESTful Web Services eli REST-ohjelmointirajapinnan toteutuksen Java-teknologialla.

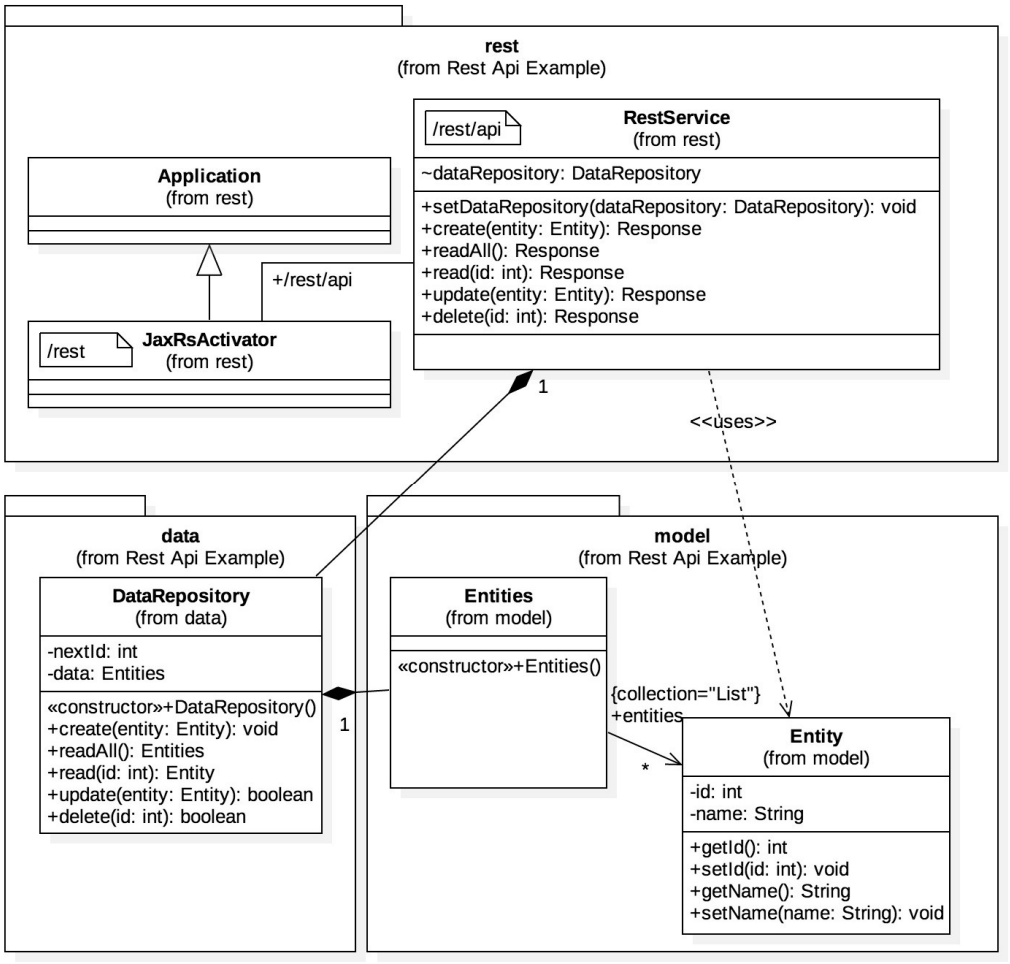
JSR-000339-spesifikaation tavoitteena on määritellä Java EE:hen sisällytetty ohjelmointirajapinta annotaatioiden, luokkien ja rajapintojen avulla, mikä mahdollistaa Java-olioiden käsittelyn suoraan rajapinnasta. Toteutuksen tulee myös olla HTTP-keskeinen: sen tulee tarjota korkean tason (high level) tuki ohjelmointirajapinnasta palveluiden käytettävyyteen standardi-HTTP-kutsujen avulla. Sisällön (content-type) kannalta toteutuksen tulee olla riippumaton esitysmuodosta ja tarjota työkalut erinäisten sisältötyyppien määrittelyyn. Tämän lisäksi spesifikaation mukaisesti kehitettyjen sovellusten tulee olla riippumattomia siitä, millä sovelluspalvelimella niitä suoritetaan, kunhan sovelluspalvelu noudattaa spesifikaatiota. (13, s. 2.)

JAX-RS 2.0 on osana JSR-000342-spesifikaatiota, joka määrittelee Java Platform, Enterprise Editionin eli Java EE:n. Java EE-spesifikaatiossa määritellään, että täydessä Java EE -tuotteessa on tuettava JAX-RS-teknologiaa. (14, s. 180.)

JAX-RS:stä on useita implementaatioita, esimerkiksi Jersey ja RESTeasy. Jersey on Oraclen ylläpitämä avoimen lähdekoodin JAX-RS-referenssi-implementaatio, kun taas RESTEasy on JBossin vastaava toteutus. Toteutusten tavoitteena on helpottaa REST-rajapintojen sovelluskehitystyötä Java-teknologialla.

2.4 Java EE REST API -esimerkki

Koodiesimerkeissä 1 ja 2 kuvattiin rakenteellista tietoa, jota koodiesimerkeissä 6–8 esiteltävä esimerkkisovellus käyttää. JSON- ja XML-vastaukset koodiesimerkeissä 3 ja 4 on luotu kuvassa 1 esiteltävällä REST API -sovellusesimerkillä. Todellisesta tehtävänannosta poiketen tiedon tallennus- ja lukuoperaatiot on toteutettu muistinvaraisella tietosäilöllä tietokannan sijaan esimerkin yksinkertaistamiseksi.



Kuva 1. Java EE REST API -esimerkin luokkakaavio.

Kuvassa 1 on kuvattu luokkakaaviona koodiesimerkeissä 1, 2, 6 ja 8 esiteltävä esimerkkisovellus. Kaaviosta ilmenee toteutettujen luokkien välinen riippuvuus. JAX-RS-sovellus vastaa web-sovelluksen URL:n komponentissa /rest, ja rajapinnan palvelun toteuttava RestService-luokka vastaa komponentissa /rest/api. Sovelluksen käyttöohjeet ja projektimalli ovat liitteissä 3 ja 4.

```
@ApplicationPath("/rest")
public class JaxRsActivator extends Application{
}
```

Koodiesimerkki 6. JaxRsActivator.java-luokka, joka käynnistää REST-rajapintakyselyt vastamaan web-sovelluksen URL:n polussa /rest. Ohjelmakoodi on kokonaisuudessaan liitteessä 5.

```
@Path("/api")
@RequestScoped
public class RestService {

  @Inject
  DataRepository dataRepository;

  public void setDataRepository(DataRepository dataRepository) {
    this.dataRepository = dataRepository;
  }

  @POST
  @Consumes({MediaType.APPLICATION_JSON,
    MediaType.APPLICATION_XML})
  public Response create(Entity entity)
  {
    dataRepository.create(entity);
    return Response.created(null).build();
  }

  @GET
  @Produces({MediaType.APPLICATION_JSON,
    MediaType.APPLICATION_XML})
  public Response readAll()
  {
    return Response.
      ok().
      entity(dataRepository.readAll()).
      build();
  }
```

```
@GET
@Path("/{id}")
@Produces({MediaType.APPLICATION_JSON,
  MediaType.APPLICATION_XML})
public Response read(@PathParam("id") int id)
{
  Entity entity = dataRepository.read(id);
  if (entity == null)
    return Response.status(404).build();

  return Response
    .ok().entity(entity).build();
}

@PUT
@Consumes({MediaType.APPLICATION_JSON,
  MediaType.APPLICATION_XML})
public Response update(Entity entity)
{
  if (dataRepository.update(entity))
    return Response.ok().build();



  return Response.status(400).build();
}

@DELETE
@Path("/{id}")
@Produces({MediaType.APPLICATION_JSON,
  MediaType.APPLICATION_XML})
public Response delete(@PathParam("id") int id)
{
  if (dataRepository.delete(id))
    return Response.noContent().build();
```

```
    return Response.status(404).build();
  }
}
```

Koodiesimerkki 7. RestService.java-luokka, joka yhdistää HTTP-metodeihin CRUD-operaatiot. HTTP-metodeja vastaavat metodit vastaavat kutsuihin web-sovelluksen URL:n polussa /rest/api. Ohjelmakoodi on kokonaisuudessaan liitteessä 6.

```
@ApplicationScoped
public class DataRepository {
  private int nextId;
  private Entities data;

  public DataRepository()
  {
    data = new Entities();
    nextId = 0;
  }

  public void create(Entity entity)
  {
    entity.setId(nextId);
    data.entities.add(entity);
    nextId++;

  }

  public Entities readAll()
  {
    return data;
  }

  public Entity read(int id)
  {
    Entity result = null;
    for (Entity entity : data.entities)
    {
      if (entity.getId() == id)
```

```
      {
        result = entity;
        break;
      }
    }
    return result;
  }

  public boolean update(Entity entity)
  {
    for (int i=0;i<data.entities.size();i++)
    {
      if (data.entities.get(i).getId() == entity.getId())
      {
        data.entities.get(i).setName(entity.getName());
        return true;
      }
    }
    return false;
  }

  public boolean delete(int id)
  {
    for (int i=0;i<data.entities.size();i++)
    {
      if (data.entities.get(i).getId() == id)
      {
        data.entities.remove(i);
        return true;
      }
    }
    return false;
  }
}
```

Koodiesimerkki 8. DataRepository.java-luokka, muistinvarainen tietovaranto, joka toteuttaa varsinaiset CRUD-operaatiot. Ohjelmakoodi on kokonaisuudessaan liitteessä 7.

Koodiesimerkkien 6–8 REST-rajapinta vastaa web-sovelluksen polussa /rest/api. @ApplicationPath-annotaatiolla merkitty JaxRsActivator-luokka saa aikaan kutsujen vastaanoton annotaatiossa merkityssä polussa. @Path-annotaatiolla merkitty RestService-luokka vastaa HTTP-kyselyihin @ApplicationPath-annotaation alla merkityssä polussa. @RequestScoped-annotaatio määrittelee, että jokaista kyselyä varten alustetaan uusi RestService-luokka, jonka tietovaranto riippuvuusinjektoidaan @Inject-annotaatiolla. Kyselyn jälkeen kyselyä varten alustettu olio tuhotaan. Kun tietovaranto taas on merkitty @ApplicationScoped-annotaatiolla, osaa Javan CDI (Contexts and Dependency Injection) injektoida oikean riippuvuuden, niin että koko ohjelman suorittamisen elinkaaren aikana RestService-luokan kaikki kutsujat käyttävät samaa instanssia tietovarannosta. Jokaista kyselyä varten on tässä toteutuksessa oma RestService-olio, mutta kaikilla RestService-olioilla on yhteinen DataRepository-olio.

@POST-, @GET-, @PUT- ja @DELETE-annotaatioilla merkityt metodit RestService-luokassa vastaavat samannimisiin HTTP-kutsuihin toteuttaen CRUD-operaatiot tietovarannolle. @Consumes- ja @Produces-annotaatioilla voidaan määritellä, minkälaista sisältöä metodiin voi syöttää (Consumes) ja mitä se voi palauttaa (Produces), esimerkiksi application/json ja application/xml. Metodit palauttavat Response-olion, jotka sisältävät HTTP-statuskoodin ja mahdollisesti palautettavan entiteetin Response-Bodyssä. Taulukossa 2 on koostettu esimerkkejä HTTP-kutsuista ja HTTP-vastauksista.

Taulukko 2. Esimerkkejä RestService-luokan HTTP-kutsuista, joilla muun muassa voidaan luoda koodiesimerkkien 3 ja 4 JSON- ja XML-muodostukset. Kyselyt ja vastaukset on toteutettu esitetyssä järjestyksessä.

HTTP-Metodi ja Header	**URL**	**Body (Kysely)**	**Vastauskoodi**	**Body (Vastaus)**

POST
Content-Type: application/json
	/rest/api	{
		"id": 0,
		"name": "Matti Meikäläinen"
		}
			201 Created	(tyhjä)

POST
Content-Type: application/json
	/rest/api	{
		"id": 0,
		"name": "Maija Meikäläinen"
		}
			201 Created	(tyhjä)

GET Accept: application/json	/rest/api		200 Ok	{ "Entity": [{ "id": 0, "name": "Matti Meikäläinen" }, {

				"id": 1, "name": "Tanja Teikäläinen" }] }
GET Accept: application/xml	/rest/api		200 Ok	<?xml version="1.0" encoding="UTF-8" standalone="yes"?> <Entities> <Entity> <id>0</id> <name>Matti Meikäläi-nen</name> </Entity> <Entity> <id>1</id> <name>Tanja Teikäläinen</nimi> </Entity> </Entities>

GET
Accept: applica-tion/json
	/rest/api/1		200 OK	{
				"id": 1,
				"name": "Tanja Teikäläinen"
				}

GET Accept: applica-tion/json	/rest/api/33		404 Not Found	

PUT
Content-Type: applica-tion/json
	/rest/api	{
		"id": 1,
		"name": "Tanja-Maija Meikäläinen"
		}
			200 OK	

GET
Accept: applica-tion/json
	/rest/api		200 OK	{
				"Entity": [
				{
				"id": 0,
				"name": "Matti Meikäläinen"
				},
				{
				"id": 1,
				"name": "Tanja-Maija Meikäläinen"
				}
				]
				}

PUT
Content-Type: applica-tion/json
	/rest/api	{
		"id": 55,
		"name": "Tanja-Maija Teikäläinen"
		}
			400 Bad Re-quest	

DELETE	/rest/api/0		204 No Con-tent	
DELETE	/rest/api/0		404 Not Found	

GET
Accept:
application/json
	/rest/api		200 OK	{
				"Entity": [
				{
				"id": 1,
				"name": "Tanja-Maija Meikäläinen"
				}
				]
				}

POST Content-Type: application/xml	/rest/api	<Entity> <id>1</id> <name>Tanja-Maija Meikäläinen</name> </Entity>	200 OK	

POST
Content-Type:
application/json
	/rest/api	{
		"virheellinen" : 1,
		}
			400 Bad Request	(Virheilmoitus)

Taulukossa 2 on kuvattu valtaosa HTTP-kyselyistä ja eri vastausvaihtoehdoista, joita koodiesimerkissä 6–8 kuvattu REST-rajapinta toteuttaa. Koska kyseessä on esimerkki, ei rajapintaa ole hiottu loppuun asti sietämään esimerkiksi virheellisiä syötteitä, vaan käyttäjälle saattaisi hyvin nopeasti paljastua vastauksia HTTP-statuskoodilla 500 - Internal Server Error, joka yleensä kuvaa tilannetta, että sovellus on käynyt odottamattomassa tilassa. Käynyt-sana sopii melko hyvin tilaton-sanan asiayhteyteen REST-rajapinnoissa. Kahden keskustelevan järjestelmän ei tarvitse olla tietoisia toistensa tilasta, vaikka sinne välillä menisi vääriä kyselyitä, vaan palvelin voi vastata kykynsä mukaan HTTP-statuskoodeilla tai pyydetyllä sisällöllä.

3 Yksikkötestaus

Ohjelmistokehitystyön yksikkötestauksen tarkoituksena on kokeilla, toimiiko kirjoitettu ohjelmakoodi oikein. Testauksen ja hyväksytyn tuloksen avulla saadaan varmuus siitä, että kehitystyössä on turvallista edetä. (15, luku 1, osio 2, kappale 1.)

Yksikkötestillä tarkoitetaan erityistä ohjelmakoodia, joka testaa ohjelmakoodissa olevan metodin tai funktion. Tätä testattavaa metodia tai funktiota kutsutaan SUT:ksi (System Under Test), ja yksi yksikkötesti testaa yhden annetun syötteen ja syötteen aiheuttaman oletetun tuloksen. (16, s. 4.)

Yksikkötesteissä annetaan erilaisia syötteitä testattavalle yksikölle, ja testissä todetaan, että annetulle syötteelle saadaan oikeanlainen vastaus. Yksikkötesteillä testataan myös yksikön kyky toimia oikein tilanteessa, jossa se saa vääränlaisen syötteen. Tämän avulla saadaan tietoon ja aikaisessa vaiheessa korjatuksi mahdolliset ongelmat, jotka voivat johtaa koko ohjelman toimimattomuuteen tuotantotilanteessa, tai jopa sisään rakentuvia tietoturvaongelmia.

3.1 Yksikkötestien toteuttaminen

3.1.1 Yksikkötestit

Yksikkötestit toteutetaan omina automaattisesti ajettavina ohjelmapätkinä yksikkötestauskirjaston avulla ja erillään tuotantokoodista. Ennen testien kirjoittamista voidaan suunnitella etukäteen, mitä arvoja yksikölle annetaan ja mitä oletetaan ulostuloarvoksi kussakin tilanteessa. Tällä tavalla voidaan hahmottaa, mitä testataan ja mitä ei.

Yksikkötestit tulee myös toistettavuuden takia pystyä ajaa automatisoidusti (vrt. nappia painamalla), minkä jälkeen kehitysympäristössä on nähtävissä ajon jälkeinen testiraportti (testit, jotka läpäistiin, ja testit, joita ei läpäisty ja niiden ulostuloarvojen ristiriidat oletamien kanssa).

Kun yksikkötestien lähdekoodi pidetään riittävän yksinkertaisena ja luettavana, voi lähdekoodi itsessään toimia testidokumenttina, josta ilmenevät testattavat syötteet ja oletetut ulostuloarvot. Yksikkötestit seuraavat ohjelmiston lähdekoodin mukana.

Edellä mainitut asiat tukeutuvat Roy Osheroven ideaalisen yksikkötestin määritelmään. (16, s. 6.)

Yksikkötestit osana ohjelmiston lähdekoodia (tuotantokoodista eristettynä), automatisoitu ajo ja visuaalinen raportointi taas ovat integroituna osana useassa ohjelmistokehitysympäristössä, kuten esimerkiksi Eclipse tai Netbeans. Yksikkötestejä voi myös ajaa komentoriviltä soveltuvilla työkaluilla, esimerkiksi Mavenilla.

3.1.2 Black box-, White box- ja Gray box -periaatteet

Yksittäisten yksikkötestien tai testien tekemistä voi lähestyä Black box-, White box- tai Gray box -periaatteella.

Black box -periaatteella tarkoitetaan tilannetta, jossa testattavana on musta laatikko. Testin tekijän tiedossa ei ole, miten musta laatikko sisäisesti toimii, vaan hän antaa laatikolle syötteitä ja lukee ulostulojen arvot. Tämä on yleinen lähestymistapa valmiiden ohjelmistokomponenttien integraatiotestauksessa tai valmiin ohjelmiston hyväksymistestauksessa. Black box -testauksella voidaan yleensä testata ohjelmiston määritysdokumenttien mukaisuutta, vääriä tai puuttuvia funktioita, rajapinnan vikoja, tietorakenteiden virheitä, käyttäytymistä, alustusta ja sulkeutumista. (17.)

White box -periaatteella tarkoitetaan vastaavasti tilannetta, jossa testattavana on lasinen (läpinäkyvä) tai avonainen laatikko. Testit voidaan suunnitella tiedossa olevan rakenteen perusteella, niin että voidaan käydä läpi mahdolliset haarautumiset ja tilamuutokset testattavan komponentin sisällä. Tämä tilanne on yleinen ohjelmiston kehitysvaiheessa ja soveltuu täten yksikkötestaukseen, koska yksikön lähdekoodi on saatavilla. White box-testausta voidaan myös toteuttaa integraatiotestauksessa. Tällä tavalla voidaan tietoisesti suorittaa ja testata kaikki haarat ja tilat, joista Black box -tilanteessa ei olisi tietoa. (18.)

Gray box -periaatteella taas tarkoitetaan tilannetta Black boxin ja White boxin välissä. Tässä tilanteessa testit voidaan suunnitella White box -periaatteella, mutta itse testit toteutetaan Black box-periaatteella eli annettujen syötteiden ja oletettujen ulostuloarvojen ja ulospäin näkyvien tilojen todentamisella. Gray box -periaatetta voidaan käyttää periaatteessa missä tahansa testeissä, mutta yleensä ohjelmistokomponenttien keskinäisessä integraatiotestauksessa. (19.)

3.1.3 Yksikkötestaus ja ohjelmointityö

Mikäli ohjelman lähdekoodi kirjoitetaan ennen yksikkötestien kirjoittamista, voidaan joutua tilanteeseen, jossa turhaa ohjelmakoodia on kirjoitettu paljon ja jälkikäteen voi olla vaikea hahmottaa, mikä on pielessä.

Edellä mainitun tilanteen syntymistä voidaan estää TDD (Test Driven Development) -lähestymistavalla. TDD:ssä testit ohjelmoidaan ensiksi, minkä jälkeen varsinainen ohjelmakoodi kirjoitetaan ja korjataan, kunnes se läpäisee kaikki testit. Tällä tavalla voidaan minimoida turhan koodin kirjoittaminen, koska tehdään vain se, mitä tarvitaan. (15, luku 1, osio Defining Unit Testing, kappale 4.)

Kun kirjoitetaan koodia ja yksikkötestejä prosessissa, voidaan myös pian nähdä, onko ohjelmiston sisäinen rakenne suunniteltu siten, että se voidaan testata. Testit antavat oikeaa kuvaa laadusta eli siitä, kuinka hyvän testaamisen koodin rakenne mahdollistaa.

Testitapaukset voidaan johdatella esimerkiksi suunnitelmasta tai määrittelydokumentista: miten jonkin komponentin oletetaan käyttäytyvän tai niin, että kokeillaan mahdolliset variaatiot oikeita ja vääriä syötteitä. Oikeiden syötearvojen käyttäytymistä voi testata antamalla arvoja, jotka antavat mahdollisimman oikeanlaista kuvaa siitä, miten testattava komponentti kykenee toimimaan määriteltyjen raja-arvojen sisällä tai rajalla ja miten se reagoi, kun se saa syötteen raja-arvojen ulkopuolelta. Väärillä, tyhjillä ja virheellisillä syötearvoilla taas voi testata, miten komponentti reagoi ja kykenee toimimaan odottamattoman syötteen kanssa. Näin saadaan kuvaa kirjoitetun ohjelman vikasietoisuudesta ja kyvystä toimia oikein virhetilanteessa, esimerkiksi antamalla oikeanlainen virheilmoitus.

Samalla kun testejä kirjoitetaan, käy myös ilmi minkä kehitettävistä metodeista ja kentistä on oltava julkisia. Näin voidaan myös yksikkötestauksen kannalta ottaa kantaa siihen, onko riittävää näkyvyyttä luokan sisälle kattavan yksikkötestauksen toteuttamiseksi. Jos Gray box -menetelmällä ei saada riittävän hyvin yksikkötestattua näkyvien metodien ja kenttien kautta, voidaan harkita refaktorointia esimerkiksi muuttamalla näkyvyyttä, muuttamalla sisäistä rakennetta tai muuttamalla jonkin osion näkyvyyttä.

3.2 Yksikkötestauskirjastot

Vaikka yksikkötestausta ylipäätään voitaisiin toteuttaa hyvinkin monella tavalla, on valmiina olemassa useita yksikkötestauskirjastoja, jotka selkeyttävät ja standardoivat työnkulkua. Esimerkiksi Java-sovelluskehitykseen on olemassa JUnit, TestNG ja Jtest. Jtest on kaupallinen ohjelmisto, kun taas JUnit ja TestNG edustavat avoimen lähdekoodin ohjelmistoja. Koska insinöörityön toteutuksessa päädyttiin käyttämään avoimen lähdekoodin kirjastoa JUnitia, käsitellään tässä ainoastaan JUnit.

Yleisesti ottaen testauskirjastot tukevat testien merkitsemistä Java-annotaatioilla, jolloin kehitystyökalut tunnistavat ne. Lisäksi niissä on ominaisuuksia, joiden avulla kehittäjä kirjoittaa väittämiä (assertions), joita vasten testit ajetaan. Kirjastoissa voi myös olla laaja tuki erinäisten toimintojen automatisoimiseen, kuten esimerkiksi testeissä käytettävän tiedon automaattinen alustus, testien ryhmittely (Test Suite), automaattinen eri syötearvojen testaus, rinnakkaisajo ja niin edelleen. (20.)

Seuraavaksi esitetään esimerkkinä yksikkötestit aikaisemmin kuvatulle DataRepository-luokalle JUnitilla toteutettuna.

Yksikkötestaus JUnitilla

JUnitin kehitystiimi kuvaa JUnitin yksinkertaiseksi viitekehykseksi, jonka avulla voidaan kirjoittaa toistettavia testejä (20). JUnitissa on tuki useaan eri kehitysympäristöön ja ohjelmointityökaluun. JUnit on Javan suosituin yksikkötestauskirjasto (15, luku 1, osio Working with JUnit 4, kappale 1).

Yksikkötestit kirjoitetaan ohjelmakoodina erikseen tuotantokoodista. Testit kirjoitetaan metodeina, jotka sisältävät yleensä väittämiä (assertions). Ohjelmoija voi kirjoittaa väittämän, joka tarkistaa, että kaksi vertailtavaa asiaa ovat väitetyllä tavalla, esimerkiksi testattavan metodin syöte antaa väitetyn ulostulon, muussa tapauksessa testi hylätään. JUnitissa on useisiin käyttötarkoituksiin väittämä-metodeja, esimerkiksi assertNull, assertSame, assertEquals, assertTrue ja assertFalse. Jos väittämiä ei voi käyttää testimetodissa, voi testin hylätä myös fail()-metodikutsulla. (21.)

Koodiesimerkissä 8 kirjoitetut DataRepository-luokan metodit voitaisiin yksikkötestata JUnit-kirjastolla seuraavaksi esiteltävän koodiesimerkin 9 avulla.

```
public class DataRepositoryJUnitTest {
  DataRepository testDataRepository;
  Entity testEntity1;
  Entity testEntity2;

final static int TEST_ENTITY1_ID = 123;
final static String TEST_ENTITY1_NAME = "Test Name1";
final static int TEST_ENTITY2_ID = 456;
```

```
final static String TEST_ENTITY2_NAME = "Test Name2";
final static String TEST_ENTITY3_NAME = "Modified Name";

@BeforeClass
public static void setUpBeforeClass() throws Exception {
}

@AfterClass
public static void tearDownAfterClass() throws Exception {
}

@Before
public void setUp() throws Exception {
  testDataRepository = new DataRepository();
  testEntity1 = new Entity();
  testEntity1.setId(TEST_ENTITY1_ID);
  testEntity1.setName(TEST_ENTITY1_NAME);
  testEntity2 = new Entity();
  testEntity2.setId(TEST_ENTITY2_ID);
  testEntity2.setName(TEST_ENTITY2_NAME);
}

@After
public void tearDown() throws Exception {
  testDataRepository = null;
  testEntity1 = null;
  testEntity2 = null;
}

  @Test
  public void testDataRepositoryConstructorConstructsEmptyCon
tainerNotNull() {

    assertNotNull("DataRepository was not constructed cor-
rectly",
```

```
      testDataRepository.readAll());
  }

  @Test
  public void testDataRepositoryEmptyAfterConstruction()
  {
    assertEquals("The repository was not empty after construc-
tion", 0, testDataRepository.readAll().entities.size());
  }

  @Test
  public void testDataRepositoryCreateAndVerifyAmountOfEnti-
ties()
  {
    assertEquals("The repository was not empty after construc-
tion", 0, testDataRepository.readAll().entities.size());

    testDataRepository.create(testEntity1);
    assertEquals("Wrong amount of entities in repository", 1,
      testDataRepository.readAll().entities.size());

    testDataRepository.create(testEntity2);
    AssertEquals("Wrong amount of entities in repository", 2,
      testDataRepository.readAll().entities.size());
}

@Test
public void testCheckCreatedDataWasCorrect() {
  testDataRepository.create(testEntity1);
  testDataRepository.create(testEntity2);
  assertEquals("There was wrong amount of entities in the re
    pository", 2, testDataRepository.readAll().entities.size());

  assertEquals("The entity contained incorrect data",
    TEST_ENTITY1_NAME,
    testDataRepository.readAll().entities.get(0).getName());
```

```
  assertEquals("The entity contained incorrect data",
    TEST_ENTITY2_NAME,
    testDataRepository.readAll().entities.get(1).getName());
  }

/* esimerkkiä lyhennetty */

  @Test(expected=NullPointerException.class)
  public void testThrowNullPointerExceptionWhenUsingGetNameOn
NullEntityInEmptyRepository() {

  @SuppressWarnings("unused")
  String testTheNameReadFromEmptyEntityInEmptyRepository =
    testDataRepository.read(0).getName();
  }

/* esimerkkiä lyhennetty */

@Test
  public void testCreateNullObject()
  {
    testDataRepository.create(null);
  }
}
```

Koodiesimerkki 9. DataRepository-luokan yksikkötestaus JUnit-kirjastolla. Ohjelmakoodi on kokonaisuudessaan liitteessä 8.

Kuten koodiesimerkissä 9 käy ilmi, JUnitia ohjataan annotaatioiden avulla. Ennen luokan rakentamista ja testien ajamista, JUnit kutsuu @BeforeClass-metodia. Testien ajojen jälkeen JUnit kutsuu @AfterClass-metodia. Nämä metodit voivat sisältää sellaisia toimenpiteitä, jotka tehdään vain kerran, esimerkiksi luodaan tietokantayhteys tai suljetaan se. Jokainen @Test-annotaatiolla merkitty metodi on erillinen testi. Ennen jokaisen testin ajoa JUnit kutsuu @Before-metodia ja jokaisen testin jälkeen @After-metodia. Edellä esitetyssä esimerkissä @Before-metodi rakentaa uuden tietovarannon ja testidatan testin käyttöön ja @After-metodi tyhjentää käytetyt tiedot pois, niin että seuraavaa testiä

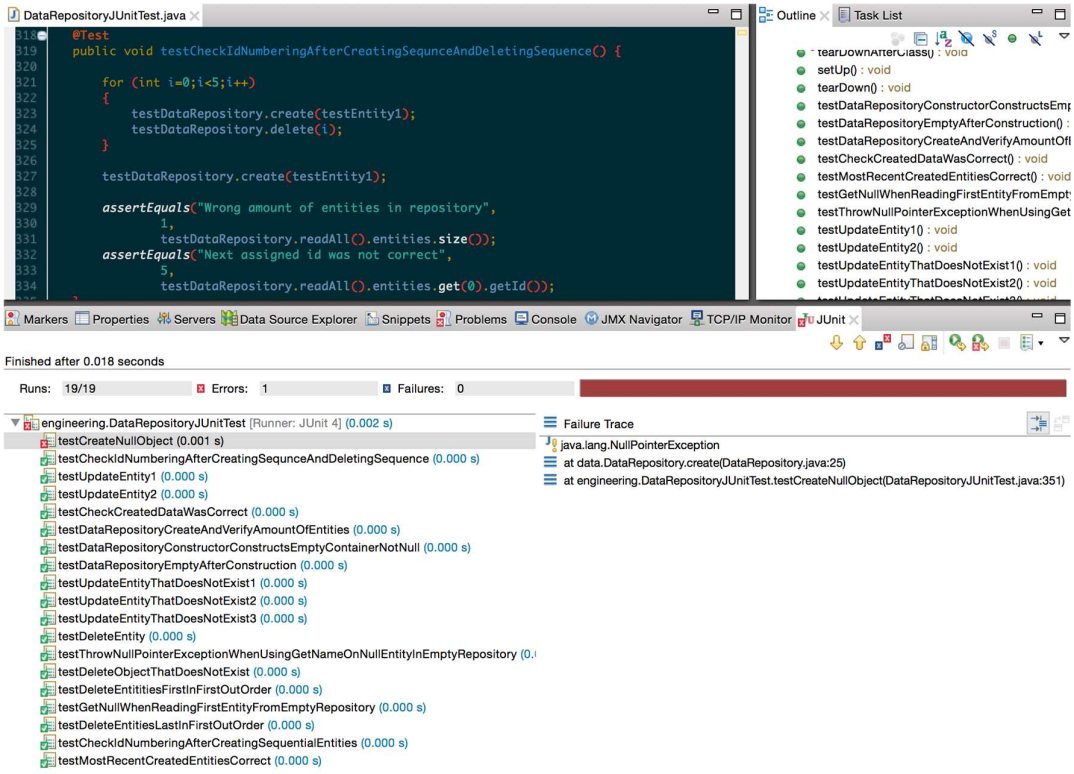
päästään suorittamaan puhtaalta pöydältä. Tämä tukee myös ajatusta siitä, että yksikkötestien onnistuminen tässä tapauksessa ei riipu testien suoritusjärjestyksestä.

@Test-annotaatiolla merkitty metodi sisältää lyhimmillään yhden assert-metodin kutsun. Assert testaa ohjelmoijan kirjoittaman väittämän ja välittää testiraporttiin siihen määritellyn virheilmoituksen. Esimerkiksi assertNotNull("DataRepository was not constructed correctly", testDataRepository.readAll()) -metodikutsu tarkistaa ettei väittämä testDataRepository.readAll() ei palauta null-oliota. Mikäli metodikutsu palauttaa null-olion, kirjaa JUnit epäonnistumisen ja välittää ”DataRepository was not constructed correctly” -virheilmoituksen käyttöliittymään tai testiraporttiin. Mikäli väittämät pitävät paikkansa, jatketaan testimetodin suoritusta ja se saa hyväksytyn arvon testiraporttiin tai käyttöliittymään.

Koodiesimerkissä 9 suoritetaan yksi testi, jonka avulla todetaan, että tietyssä virhetilanteessa ohjelman tulee antaa NullPointerException-poikkeus. Tätä ei voida kirjoittaa väittämällä, vaan testimetodi merkitään @Test(expected=NullPointerException.class)-annotaatiolla.

Esimerkin vuoksi testCreateNullObject-metodi on testi, joka epäonnistuu, ja sillä saadaan kiinni ohjelmointivirhe DataRepository-luokassa. Liitteessä 8 metodilla on myös @Ignore-annotaatio, joka kertoo JUnitille, että testiä ei ajeta. Näin testin olemassaolo huomioidaan raporteissa, eikä niitä tarvitse kommentoida pois, jos niitä ei ajeta.

Kun JUnit-yksikkötestit ajetaan kehitysympäristössä, jossa on tuki JUnit-yksikkötestien integraatiolle, saadaan käyttöliittymään suoraa palautetta testien ajosta. Kuvassa 2 on esimerkkinäkymä siitä, miten ajetut yksikkötestit on onnistuneet. Koodiesimerkissä 9 on havainnollistettu onnistuvia testejä ja yksi testi, joka paljastaa kirjoitetun ohjelmakoodin virheen ja sovelluskehittäjälle välittyvän virheilmoituksen.



Kuva 2. Graafinen näkymä Eclipse-kehitysympäristön JUnit-integraatiosta, testiraportista ja virheilmoituksesta.

3.3 Riippuvuuksia sisältävien komponenttien yksikkötestaus

Usein liike-elämän ohjelmistot ovat monimutkaisia rakennelmia. Järjestelmissä on useita alijärjestelmiä, rajapintoja, tietokantayhteyksiä ja muita riippuvuuksia sisältäviä komponentteja. Koska yksikkötestit itsessään testaavat pienimpiä testattavia yksiköitä, eli esimerkiksi luokkia ja niiden metodeja, voidaan vahingossa ajautua integraatiotestauksen puolelle, kun testiin sisältyy riippuvuuksia muihin komponentteihin. Joidenkin toiminnallisuuksien testauksessa tämä voi olla ainoa tapa. Joka tapauksessa osana yksikkötestausta pitäisi voida todeta, miten jokin luokka ja sen metodit toimivat itsekseen saadessaan erilaisia syötteitä ja epäsyötteitä. Tämä kysymys olisi vastassa esimerkiksi tilanteessa, jossa kirjoitettaisiin yksikkötestit koodiesimerkkien 7 ja 8 REST-rajapinnalle ja tietovarannolle ja kehityskaaren myöhemmässä integraatiotestausvaiheessa olisi tarkoitus testata järjestelmien tai komponenttien välisiä integraatioita.

3.3.1 Lähestymistapana integraatiotestaus

Koska REST-rajapinnalla ja sen käyttämällä tietovarannolla on integraatiosuhde, olisi luonnollista lähestyä ongelmaa suoraan integraatiotestausnäkökulmasta. Yksi vaihtoehto voisi olla kirjoittaa REST-asiakassovellus, joka muodostaisi kyselyitä kehitysympäristössä olevaan järjestelmään. Jos integraatiotestaus toteutettaisiin koodiesimerkkien 7 ja 8 rajapinnalle ja tietovarannoille, voitaisiin luoda kyselyitä, jotka luovat tietueita tietovarantoon ja lukevat kirjoitettuja tietoja sieltä. Yksikkötestit tehtäisiin Gray box -periaatteella, koska tiedettäisiin sisäinen rakenne mutta ainoa näkyvä rajapinta on sama, mikä näkyy tosielämässä toiselle järjestelmälle. Ongelmaksi voisi muodostua se, että valittu testaustapa sisältää paljon hallitsemattomia riippuvuuksia, kuten esimerkiksi tietokantayhteydet, tiedostojärjestelmät, verkkoyhteydet, palvelimet, ympäristö ja niin edelleen.

3.3.2 Arquillian-kirjaston käyttö integraatiotestauksessa

JBossin (Red Hat) avoimen lähdekoodin Arquillian-testikirjasto pyrkii ratkaisemaan tyypilliset ongelmat, joihin törmätään integraatiotesteissä JavaEE-sovelluksissa. Arquillianin menetelmä perustuu siihen, että testejä varten rakennetaan paketti, joka sisältää kaikki testin tarvitsemat riippuvuudet. Testit ajetaan paketoinnissa, joka käynnistetään sovelluspalvelimessa pelkästään testien ajoa varten, tai vaihtoehtoisesti testit voidaan ajaa asiakas-tilassa paketoinnin ulkopuolella. Näin integraatiotestejä varten saadaan suljettu ympäristö, joka vastaa tuotantoympäristöä, ja integraatiotestit voidaan konfiguroimalla tarvittaessa ajaa eri sovelluspalvelimilla. Ajon aikana testit, testiympäristö, tietokannan taulut, tiedostot ja niin edelleen paketoidaan, ajetaan, ja siivotaan pois. (22).

Etenkin kantatransaktioihin liittyvien operaatioiden testaus yksinkertaistuu Arquillianilla, koska testit eivät enää oleta tietokannalta tai sen tauluilta tiettyä tilaa, vaan tiedot kirjoitetaan testiä varten tauluun ja testin jälkeen taulut puretaan (23).

Arquillian käyttää testiajurin pohjalla JUnitia, joten yksikkötestit voidaan siirtää Arquillian-testeiksi pienellä vaivalla. Merkittävä ero JUnit-yksikkötestien ja Arquillian-testien välillä on Arquillian-testien paketointi, joka konfiguroidaan @Deployment-annotaatiolla. Paketointiin konfiguroidaan luokat, resurssit ja muut riippuvuudet, joita tarvitaan. Riippuen testattavan ohjelman kompleksisuudesta kasvaa Arquillian-testien paketoinnin konfiguroinnin kompleksisuus samassa suhteessa. Testit itsessään pysyvät yksinkertaisina, myös siksi, ettei niissä käytetä mock-olioita tai muita vastaavia menetelmiä, vaan varsinaista

ohjelmaa. Arquillian-testiajon paketin jakelua varten tarvitaan arquillian.xml-konfiguraatiotiedosto, joka on kuvattu liitteessä 9.

JUnit-yksikkötestit voidaan ajaa Arquillianilla hyvin pienellä muutoksella, kuten koodiesimerkissä 10 havainnollistetaan.

```
@RunWith(Arquillian.class)
public class DataRepositoryArquillianTest {
  DataRepository testDataRepository;
  Entity testEntity1;
  Entity testEntity2;

  /* Esimerkkiä lyhennetty */
}
```

Koodiesimerkki 10. DataRepository-luokan yksikkötestaus JUnit-kirjastolla, mutta käyttäen Arquillianin testiajuria. Ohjelmakoodi on kokonaisuudessaan liitteessä 10.

Kuten koodiesimerkissä 10 käy ilmi, riittää @RunWith(Arquillian.class)-annotaatio tiedoksi JUnitille, että testiajossa käytetään Arquillianin testiajuria. Koodiesimerkissä 11 havainnollistetaan RestService-luokan testit integraatiotestinäkökulmasta Arquillianilla toteutettuna.

```
@RunWith(Arquillian.class)
public class RestServiceArquillianTest {
  final static String TEST_ENTITY1_NAME = "Test Name1";
  final static int TEST_ENTITY1_ID = 0;
  /* Esimerkkiä lyhennetty */

  DataRepository testDataRepository;

  @Inject
  RestService testRestService;

  @Deployment
  public static WebArchive createDeployment() {
    return ShrinkWrap.create(WebArchive.class)
```

```
    .addClasses(JaxRsActivator.class, RestService.class,
      DataRepository.class, Entity.class, Entities.class);
}

@BeforeClass
public static void setUpBeforeClass() throws Exception {
}

@AfterClass
public static void tearDownAfterClass() throws Exception {
}

@Before
public void setUp() throws Exception {
  testDataRepository = new DataRepository();
  testRestService.setDataRepository(testDataRepository);

  testEntity1 = new Entity();
  testEntity1.setName(TEST_ENTITY1_NAME);
  testEntity1.setId(TEST_ENTITY1_ID);

  testEntity2 = new Entity();
  testEntity2.setName(TEST_ENTITY2_NAME);
  testEntity2.setId(TEST_ENTITY2_ID);
}

@After
public void tearDown() throws Exception {
  testDataRepository = null;
  testEntity1 = null;
  testEntity2 = null;
}

@Test
public void testRestServiceReadAllStatus200Ok() {
  Response testResponse;
```

```
  testResponse = testRestService.readAll();
  assertEquals("Status was not 200 Ok", 200,
    testResponse.getStatus());
}

/* Koodiesimerkkiä lyhennetty */

@Test
public void testRestServiceCreateOneReaditWithReadAllMethod()
{
  Response testResponse;

  testResponse = testRestService.create(testEntity1);
  assertEquals("Status was not 201 Created", 201,
    testResponse.getStatus());

  testResponse = null;

  testResponse = testRestService.readAll();

  Entities testResponseEntities =
    (Entities) testRespose.getEntity();
  assertEquals("Wrong amount of entities in response", 1,
    testResponseEntities.entities.size());
  assertEquals("Wrong id", TEST_ENTITY1_ID,
    testResponseEntities.entities.get(0).getId());
  assertEquals("Wrong name", TEST_ENTITY1_NAME,
    testResponseEntities.entities.get(0).getName());
  assertEquals("Status was not 200 OK", 200,
    testResponse.getStatus());
}

/* Koodiesimerkkiä lyhennetty */


@Test
```

```
  public void testRestServiceCreateEntityNullObject() {
    Response testResponse;
    testResponse = testRestService.create(null);
    assertEquals("Status was not 201 Created", 201,
      testResponse.getStatus());
  }

  /* Koodiesimerkkiä lyhennetty */

}
```

Koodiesimerkki 11. RestService-luokan yksikkötestit suoritettuna integraatiotestimenetelmällä käyttäen Arquilliania. Ohjelmakoodi on kokonaisuudessaan liitteessä 11.

Kuten koodiesimerkistä 11 ilmenee, @Deployment-annotoitu testin paketointi lisää tässä metodikutsussa pakettiin kaikki luokat, joita tarvitaan testin kokonaisuuden ajamisen aikana. Näin kaikki oikeat riippuvuudet ovat saatavilla testin ajon aikana. Testien ajon aikana testataan koko ketjun riippuvuuksia. Huomioitavaa on, että @Inject-annotaatiolla riippuvuusinjektoidaan @RequestScoped-annotoitu RestService-olio ja testien ajojen välillä testDataRepository-olio alustetaan tyhjästä, koska muuten @ApplicationScoped-annotoitu DataRepository-olio käyttäisi testistä toiseen samaa tietovarantoa. Muuten ei voitaisi taata testien satunnaista ajojärjestystä, koska testit on tässä tapauksessa suunniteltu siten, että ne ajetaan aina tyhjää tietovarantoa vasten.

Koodiesimerkissä kirjoitettu testi, testRestServiceCreateEntityNullObject-metodi, tuo ilmi yhden integraatiotesti-lähestymistavan edun. Tällä testillä saadaan kiinni toteutetun ohjelman heikkous, joka sallii null-olion antamisen parametrina RestService.create-metodille aiheuttaen NullPointerException-poikkeuksen DataRepository.create-metodissa. Vastaavaa virhettä ei saada kiinni, kun luvussa 3.3.3 käsitellään mock-olioihin perustuva testauslähestymistapa, mutta asia tulisi korjata DataRepository-luokassa, jonka testit voidaan todeta tämän perusteella puutteellisiksi.

3.3.3 Lähestymistapana mock-oliot

Vaihtoehtona integraatiotestaukselle, jossa testataan ohjelmakoodi oikeiden tai vastaavien riippuvuuksien kanssa, voidaan käyttää riippuvuuksiin viittaavien olioiden tilalla

mock-olioita. Englannin kielen mock-sanalla tarkoitetaan jäljitelmää tai matkijaa. Mock-oliot ovat käyttökelpoisia seuraavissa tilanteissa:

- Riippuvuudet eivät ole saatavilla, esimerkiksi testiympäristössä.
- Riippuvuuden käyttö on resurssimielessä kallista, esimerkiksi raskas tietokantatransaktio.
- Oikean riippuvuuden käyttö tekisi todellisuudessa jotain, mitä ei haluta testiajossa tapahtuvan, esimerkiksi testidatan kirjoitus tuotantokantaan.

Mastering Unit Testing Using Mockito and JUnit -kirjassaan Sujoy Acharya kuvaa mock-oliot ja muut sensukuiset menetelmät (test doubles) elokuvan sijaisnäyttelijöinä, jotka näyttelevät vaaralliset kohtaukset tai esiintyvät näyttelijän puolesta. (15, luku 3, kappale 2.) Nämä test double -oliot ovat seuraavat:

- Dummy – vain testin ajoa varten rakennettu testattavan yksikön käyttämä olio, joka ei tee mitään, mutta suojaa esimerkiksi NullPointerException-poikkeukselta.
- Stub – varsinaisen riippuvuuden sijasta käytettävä testiajoa varten rakennettu luokka, joka on ohjelmoitu toimimaan tietyn olettamuksen mukaan, jotta tietyssä testissä riippuvuudesta saadaan tietty vastaus.
- Fake – varsinaisen riippuvuuden sijasta käytetty aliluokka, jossa hankala metodi on ylikirjoitettu toimimaan testin kannalta jouhevammin.
- Mock – varsinaisen riippuvuuden sijasta käytetty jäljitelmä, jolle on määritelty tietyt palautusarvot, kun sen metodeja kutsutaan ja pitää kirjaa kutsujen määristä. Jäljitelmä ei itsessään toteuta mitään ohjelmakoodia, mitä jäljitelty riippuvuus toteuttaa.
- Spy – riippuvuudeksi naamioitu luokka, joka välittää kutsun oikealle riippuvuudelle ja pitää kirjaa, että sen metodeja kutsutaan.

(15, luku 3.)

Kun yksikkötestissä halutaan käyttää muita kuin niiden kovakoodattuja riippuvuuksia, tulisi luokkiin toteuttaa rakentimet, joiden avulla voidaan riippuvuusinjektoida yksikkötesteissä haluttu test double -olio. Yleensä tuotantokoodin eheyden takia oletusrakennin pidetään sovellusta palvelevana ja test double -olioiden riippuvuusinjektointia varten tehdään erikseen rakentimet. (15, luku 9, osio Identifying constructor issues.)

Koodiesimerkissä 12 kuvataan RestService-luokan yksikkötestit toteutettuna JUnitilla ja Mockito-kirjastolla ilman integraatiotestiriippuvuuksia.

```
public class RestServiceJUnitTest {
  RestService testRestService;
  DataRepository mockDataRepository;

  Entities testEntitiesWithOneEntity;
  Entity testEntity1, testEntity2;

  final static String TEST_ENTITY1_NAME = "Test Nimi1";
  final static int TEST_ENTITY1_ID = 0;

  final static String TEST_ENTITY2_NAME = "Test Nimi2";
  final static int TEST_ENTITY2_ID = 1;

  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
  }

  @AfterClass
  public static void tearDownAfterClass() throws Exception {
  }

  @Before
  public void setUp() throws Exception {
    testEntitiesWithOneEntity = new Entities();
    testEntity1 = new Entity();
    testEntity1.setName(TEST_ENTITY1_NAME);
    testEntity1.setId(TEST_ENTITY1_ID);

    testEntitiesWithOneEntity.entities.add(testEntity1);

    testEntity2 = new Entity();
    testEntity2.setName(TEST_ENTITY2_NAME);
    testEntity2.setId(TEST_ENTITY2_ID);

    mockDataRepository = mock(DataRepository.class);
    when(mockDataRepository.readAll()).
```

```
    thenReturn(testEntitiesWithOneEntity);
  when(mockDataRepository.read(0)).
    thenReturn(testEntity1);
  when(mockDataRepository.read(1)).
    thenReturn(null);
  when(mockDataRepository.delete(0)).
    thenReturn(true);
  when(mockDataRepository.delete(1)).
    thenReturn(false);
  when(mockDataRepository.update(testEntity1)).
    thenReturn(true);
  when(mockDataRepository.update(testEntity2)).
    thenReturn(false);

  testRestService = new RestService();
  testRestService.setDataRepository(mockDataRepository);
}

@After
public void tearDown() throws Exception {
  testRestService = null;
  mockDataRepository = null;
  testEntitiesWithOneEntity = null;
  testEntity1 = null;
  testEntity2 = null;
}



@Test

public void testRestServiceCreateEntityStatus201Created()
  throws Exception{
  Response testResponse = testRestService.create(testEntity2);
  assertEquals("Status was not 201 Created", 201,
    testResponse.getStatus());
```

```
    verify(mockDataRepository).create(testEntity2);
  }

  @Test
  public void testRestServiceCreateNullObject() throws
    Exception{

    Response testResponse = testRestService.create(null);
    assertEquals("Status was not 201 Created", 201,
      testResponse.getStatus());
    verify(mockDataRepository).create(null);
  }

  @Test
  public void testRestServiceReadAllReturnsTestEntityInEntities-
Status200Ok() throws Exception{

    Response testResponse = testRestService.readAll();
    Entities testResponseEntities =
      (Entities) testResponse.getEntity();
    assertEquals("Wrong number of entities was returned", 1,
      testResponseEntities.entities.size());
    assertEquals("Entity had wrong id", TEST_ENTITY1_ID,
      testResponseEntities.entities.get(0).getId());
    assertEquals("Entity had wrong name", TEST_ENTITY1_NAME,
      testResponseEntities.entities.get(0).getName());
    assertEquals("Status was not 200 OK", 200,
      testResponse.getStatus());
    verify(mockDataRepository).readAll();
  }

  /* Koodiesimerkkiä lyhennetty */

}
```

Koodiesimerkki 12. RestService-luokan yksikkötestit suoritettuna JUnitilla ja mock-oliolla. Ohjelmakoodi on kokonaisuudessaan liitteessä 12.

Koodiesimerkissä 12 kuvattiin RestService-luokan testaus mock-oliolla. Huomioitavaa on, että DataRepository-luokan tilalle tehdään mock-olio, jota voidaan käyttää varsinaisen riippuvuuden ”sijaisnäyttelijänä”. Mockiton mock(DataRepository.class)-metodikutsu luo DataRepositorysta aliluokan, joka on mock-olio. Koska mock-olio itsessään ei peri pääluokan toiminnallisuutta, sille tulee määritellä toiminnallisuus, jota se toteuttaa metodikutsuille. Tämän määrittely tehdään koodiesimerkissä ilmenevällä when(mockDataRepository.readAll()).thenReturn(testEntitiesWithOneEntity)-metodikutsulla. Kun mockDataRepository-olion readAll-metodia kutsutaan, palauttaa metodikutsu testEntitiesWithOneEntity-olion.

Mock-olioita käytettäessä on huomioitava, että silloin testejä kirjoitettaessa testit eivät paljasta mitään integraatioriippuvuuksista testattavan luokan ja mock-olioksi muunnetun olion välillä, saatikka mock-olion toiminnallisuutta muuta kuin mock-oliona. Siksi when(…).thenReturn(…)-metodikutsut tulee rakentaa niin, että ne palvelevat tiettyä testiskenaariota. Koodiesimerkissä 12 on havainnollistettu testRestServiceCreateNullObject-testissä, että mock-olioilla ei saada kiinni NullPointerException-poikkeusta, joka ilmeni integraatiotesteissä ja jonka olisi pitänyt ilmetä jo DataRepository-yksikkötesteissä.

Mock-olion kutsut voi tarkistaa verify-metodilla, kuten koodiesimerkissä 12 tarkistetaan verify(mockDataRepository).readAll()-metodikutsulla, että mock-olion readAll()-metodia on kutsuttu.

Muussa tapauksessa testien kirjoitus ei merkittävästi poikkea integraatiotesteistä. Vastaavalla tavalla SetUp- ja TearDown-metodeissa alustetaan tieto joka ajoa varten. Suurena erona on testitapausten määrä, joka mock-olion avulla pienenee pelkästään RestService-luokan ydintoiminnallisuuteen eli pääasiassa väittämien statuskoodien palautukseen, koska varsinainen tiedon käsittely sijaitsee DataRepository-luokassa. Testien pituutta ja eroavaisuuksia voi verrata liitteissä 11 ja 12.

4 REST-rajapinnan ja yksikkötestien toteutus

4.1 Insinöörityön kehitystyöprosessi

Insinöörityön toteutusvaiheen työprosessina toimi tätä insinöörityötä varten sovellettu muoto Elisa Appelsiinin kehitysprosessista, joka kuitenkin noudatti niin paljon kuin mahdollista Elisa Appelsiinin sillä hetkellä voimassa olevia käytäntöjä.

Toteutetun REST-rajapinnan pohjana toimi esimerkkidokumentaatio, joka kuvasi, miten rajapinta tulisi vastaamaan kutsuihin. Kehitystyön alkaessa joitakin suunnitelmia tarkennettiin palvelemaan toteutusta, ja suunniteltu toteutus kehittyi matkan aikana alkuperäisistä suunnitelmista.

Toimeksiannot, joilla toteutus tehtiin kehitysprosessissa iteratiivisesti valmiimmaksi, kirjattiin Elisa Appelsiinin Vainukoira-järjestelmään (Atlassian Jira) tiketeiksi. Jirasta otettiin tiketti työn alle, ja etenemisestä voitiin dokumentoida tekemällä kirjauksia tai muutoksia suoraan toimeksiantoihin. Työkalun avulla projektiryhmä on ajan tasalla etenevistä töistä ja jäljellä olevasta työstä.

Kun yhden toimeksiannon koodi (ml. yksikkötestit) on kirjoitettu valmiiksi, viedään koodi versionhallintaan, joka tässä toimeksiannossa on Mercurial. Versionhallinta sisältää kaikki ohjelman elinkaaren aikana tehdyt muutokset sekä tiedon tekijästä ja muutoksen ajankohdasta. Versionhallinta mahdollistaa usean samanaikaisen kehittäjän työskentelyn saman projektin kanssa. Toteutetun REST-rajapinnan kehitystyöstä tehtiin oma haara (branch), joka myöhemmin yhdistetään projektin päähaaraan ennen tuotantoon vientiä. Yksi vienti (commit) versiohallintaan vastasi lähtökohtaisesti yhden tiketin sisältöä, ellei työtä purettu pienempiin osiin.

Viennin jälkeen tiketin suoritus meni katselmointivaiheeseen, jolloin muu projektiryhmä katselmoi työn (kirjoitetun koodin). Versiohallintaan vienti käynnisti katselmointiprosessin, jonka kehittäjä voi tehdä Elisa Appelsiinin Kalamalja-järjestelmässä (Atlassian FishEye & Crucible). Katselmointiprosessissa muut kehittäjät voivat tehdä huomautuksia koodiin. Huomautukset näkyvät projektiryhmälle ja ennen kaikkea kehittäjälle, joka suoritti viennin.

REST-rajapinnan dokumentaatio kirjattiin Elisa Appelsiinin Apaja-järjestelmään (Atlassian Confluence). Dokumentaatiossa annettiin esimerkkejä kutsuista ja oletettavista vastauksista. Lisäksi kuvattiin vastausten tietorakenne ja rakenteen tietotyypit.

Kehitystyössä keskeisesti käytetty ohjelmisto koostui seuraavista sovelluksista:

- Eclipse IDE, Java-ohjelmointiympäristö, jossa ohjelmistokehitystyö pääasiassa tapahtui
- Apache Maven, build automation -työkalu, jota käytettiin ohjelman kääntämisessä
- Mercurial, ohjelmistoversiohallinta, jota käytettiin projektin aikana versiohallintaan
- Microsoft SQL Server Management Studio, jota käytettiin testitietokannan konfiguroinnissa ja testikyselyjen suorittamisessa ja tarkistamisessa
- Postman, REST-kyselyjen rakennus- ja testaussovellus, jolla kokeiltiin REST-kyselyjä ja todennettiin vastauksia manuaalisesti
- Google Chrome, internet-selain, jolla kokeiltiin REST-kyselyjä
- Notepad++, ohjelmoijan tekstinkäsittelyohjelma, jota käytettiin muun muassa leikepöytänä REST-kyselyjen ja T-SQL-kyselyjen kirjoittamiseen.

Toteutuksessa tärkeässä roolissa käytetyt Java-kirjastot olivat

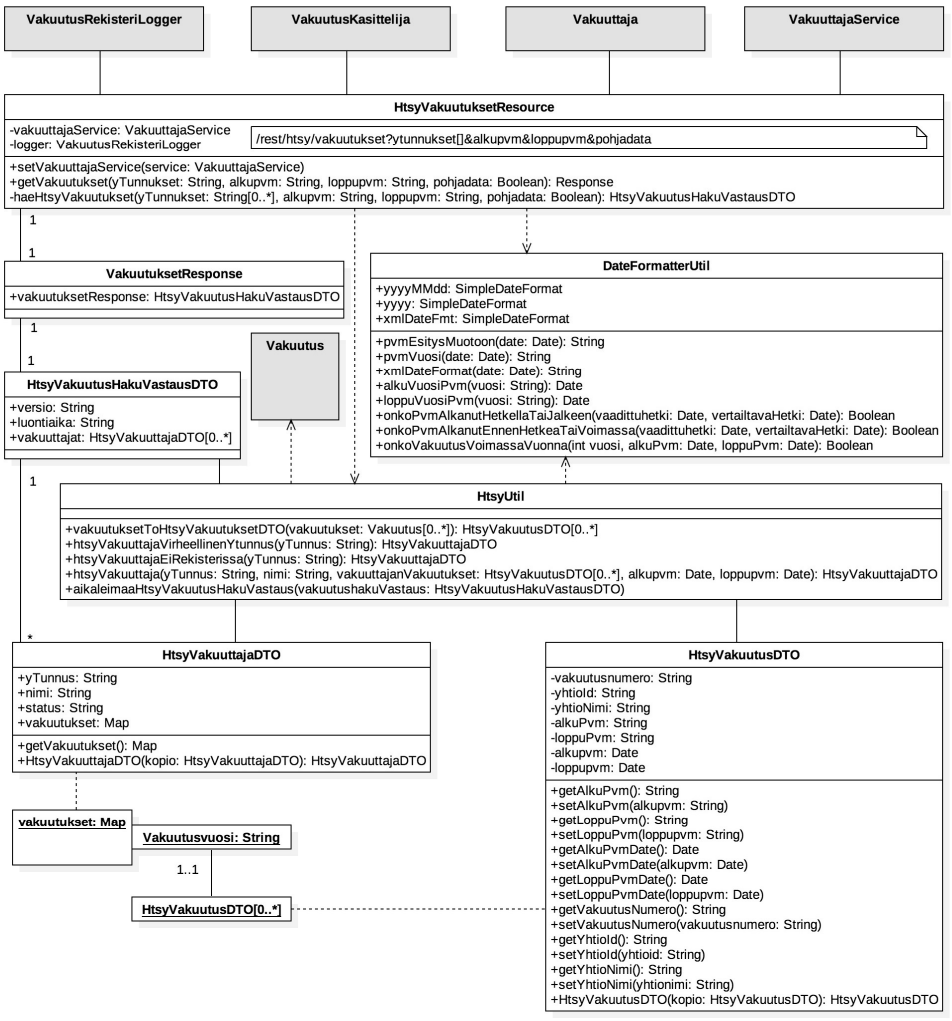
- RESTEasy, JAX-RS-implementaatio
- JUnit, yksikkötestauskirjasto
- Mockito, yksikkötestauksissa käytetty mockauskirjasto
- Java Time-, Date-, Calendar-, DateFormat- ja SimpleDateFormat-kirjastot.

Kehitysympäristö käsitteenä sisälsi useita työkaluja, joita käytettiin työnkulun eri osissa, ja niillä kaikilla oli rooli toteutuksessa, eikä pelkästään ohjelmontiympäristösovelluksella (IDE, Integrated Development Environment).

4.2 Toteutettu ohjelmakomponentti ja arkkitehtuuri

Insinöörityöprojektissa toteutettiin REST-rajapinta olemassa olevaan Java EE Web -sovellukseen. Rajapinnan tarvitsemat palvelut ja tietokantakyselyt olivat pääasiassa olemassa, poislukien pienet muokkaukset joita tehtiin versioimalla tietokantakyselyitä.

Kuvasta 3 ilmenee toteutetun kokonaisuuden rakenne riippuvuuksien kannalta sekä uuteen toteutukseen että vanhaan. Ennen insinöörityöprojektin alkua olemassa ollut toteutus on kuvattu Black box -periaatteella harmaina luokkina kaavioissa, kun taas toteutetut luokat, kentät ja metodit on kuvattu implementaatiota lukuun ottamatta Gray box -periaatteella.



Kuva 3. Insinöörityöprojektissa toteutetun REST-rajapinnan luokkakaavio.

REST-rajapinta (HtsyVakuutuksetRestResource) itsessään vastaa pelkästään HTTP:n GET-metodiin, koska palvelun tarkoituksena on hakea tietoja vakuuttajista, syötteenä taulukko Y-tunnuksia ja vapaaehtoisena parametrina aikarajaus. Syötteen tarkistus tehdään HtsyVakuutuksetRestResource-luokassa käyttäen apuna toteutetun DateFormatterUtil- ja olemassa olevan VakuutusKasittelija-luokan metodeja.

Vastauksen muodostamista varten syötetään oikean muotoisiksi Y-tunnuksiksi varmistuneet Y-tunnukset olemassa olevan VakuuttajaService-luokan metodille, joka hakee tietokannasta Vakuuttaja-luokissa esitetyt vakuutuksenottajien tiedot kyseisestä taulusta.

Vierasavaimella taas viitataan Vakuutus-luokan tauluun, jolloin saadaan vakuutuksenottajien vakuutukset. Vakuutus-olioita ja Vakuuttaja-olioita ei voida suoraan palauttaa palvelun vastauksessa sellaisenaan, ja siksi niille on toteutettu DTO (Data Transfer Object, tiedonmuunto-olio) -luokat. DTO-olioita käytetään tiedon palauttamisessa REST-rajapinnasta.

Muodostaessaan vastausta HtsyVakuutuksetRestResource rakentaa olion VakuutuksetResponse, joka sisältää HtsyVakuutusHakuVastausDTO-olion. HtsyVakuutusHakuVastausDTO sisältää aikaleiman, rajapinnan versionumeron ja HtsyVakuuttajaDTO-taulukon. HtsyVakuuttajaDTO-taulukko HtsyVakuuttajaDTO-taulukon olio sisältää vakuuttajan nimen, Y-tunnuksen, statuksen ja hajautustaulukon vakuutukset. Hajautustaulun avaimina on vakuutusvuosi, ja vakuutusvuoden arvona on HtsyVakuutusDTO-taulukko. HtsyVakuutusDTO-taulukon olio sisältää tiedot vakuutusnumerosta, vakuutusyhtiöstä ja vakuutusyhtiön id:stä ja vakuutukseen alku- ja loppupäivämäärän.

HtsyUtil-apuluokka sisältää metodit Vakuutus-olioiden muuntamisesta HtsyVakuutusDTO-luokkaan sekä toiminnallisuuden vakuuttajan statuksen kirjaamiseen kysytylle Y-tunnukselle. DateFormatterUtil-apuluokka taas sisältää työkaluja vuosien ja päivämäärien muuntamiseen merkkijonoista Date-luokkaan ja päinvastoin sekä oleellisen logiikan, jota käytetään, kun rajapinta määrittelee, onko vakuutus voimassa tiettynä vuonna. Tätä logiikkaa käytetään, kun HtsyVakuuttajaDTO-luokan hajautustaulua rakennetaan ja sijoitetaan vakuutuksia voimassaolovuosien mukaisesti hajautustaulun oikeiden avaimien alle.

Kun vastaus on rakentunut VakuutuksetResponse-olioon, se palautetaan 200 OK HTTP -statuskoodin kanssa vastauksen body-osassa JSON-koodattuna asiakkaalle. Virheellisissä tapauksissa (virheellinen syöte) palautetaan 400 Bad Request HTTP -statuskoodi ja virheilmoitus vastauksen body-osassa.

4.3 Yksikkötestit ja teknologiavalinnat

Elisa Appelsiinin aikaisemmissa Java-projekteissa, joissa yksikkötestausta on laajemmin toteutettu, on yksikkötestaus toteutettu Arquillian-testialustalla. Koska tämän insinöörityön tuotos tehtiin olemassa olevaan projektiin, jolle ei ollut tehty yksikkötestausta, to-

dettiin, että yksikkötestit voitaisiin toteuttaa kevyemmällä ratkaisulla. Yksikkötestausteknologiaksi valitun ratkaisun tulisi tukea kehitetyn ohjelmakomponentin laajuutta. Päätös, että poikettaisiin aikaisemmin käytetystä teknologiasta tässä kysymyksessä, syntyi myös aikataulullisista syistä ja siitä, että alun perin suunnitellun insinöörityön rajaus olisi laajentunut.

Alun perin tässä insinöörityösuorituksessa tehtävänantaja ei vaatinut yksikkötestausta, koska normaalikäytäntö laadunvalvonnassa oli katselmointi ja manuaalinen testaus toisen kehittäjän tai testaajan suorittamana. Normaalikäytännössä testaaja testaa Black box -periaatteella dokumentaation perusteella ja käy tarvittaessa dialogia kehittäjän kanssa.

Koska kyseessä oli insinööriopiskelijan työsuoritus, todettiin insinöörityöprojektia aloitettaessa, että insinöörityön tutkimuksen kohteeksi tarkennettaisiin REST-rajapintakehitys ja suoritetun kehitystyön tuotteen yksikkötestaus. Tällä tarkennuksella saataisiin lisää varmuutta insinöörityön toteutuksen laadusta, ja se vähentäisi tekijän katselmointiin vietävää ja palautettavaa ohjelmakoodia.

Valinta yksikkötestauskirjastojen JUnit ja TestNG välillä tehtiin nopeasti, koska todettiin, että yleisellä tasolla JUnit on laajemmin käytössä ja informaatiota tuntui olevan helpommin ja laajemmin saatavilla internetissä.

Riippuvuuksia sisältävien komponenttien yksikkötestaus päädyttiin tekemään mock-olioilla Mockito-kirjastolla, koska todettiin, että integraatiotestaus-lähestymistapa olisi aiheuttanut rajatussa ajassa turhaa työtä, kun aikaisemman kirjoitetun ohjelmakokonaisuuden laadunvalvonta oli jo suoritettu katselmoinnilla ja manuaalisella testauksella. Mockito-kirjasto valittiin saatavilla olevan dokumentaation ja olemassa olevan avoimen yhteisön ja suosion perusteella.

Integraatiotestaus-lähestymistapaa vältettiin myös siksi, että insinöörityöprojektissa kirjoitettu ohjelmakoodi käytti valmiiksi tehtyjä tietokantakyselyjä ja muita palveluita ja teki korkeintaan pieniä triviaaleja variaatioita olemassa olevista metodeista, jotka oli siis jo katselmoitu ja testattu aikaisemmassa projektin vaiheessa.

4.4 Testien kattavuus ja rajaus

Toteutetuilla yksikkötesteillä katettiin REST-rajapinnan syötteentarkastus ja HTTP-palautukset sekä statuskoodit, DateFormatterUtil-luokan päivämäärämuunnokset merkkijonoesitysmuotojen ja Date-olioiden välillä sekä DateFormatterUtil-luokan logiikka, jonka avulla määriteltiin, oliko vakuutus voimassa sinä vuotena, jota vastaavan avaimen alle vakuutus aiottiin sijoittaa vastausoliossa.

REST-rajapinnan syötteentarkistuksesta testattiin, että rajapinta reagoi toivotulla tavalla null-olioihin, päivämäärät osattiin lukea merkkijonoista, rajapinta vastasi oikein kutsuihin sallittujen syötemäärien puitteissa ja osasi palauttaa vastauksessa oikealla statuskoodilla merkittyjä vakuuttajia. Sallittua syötemäärää testattiin sekä normaali- että ääriarvoilla.

Päivämäärämuunnosten välille toteutettiin muutama tarvittava testi, joita ei monimutkaistettu siitä syystä, että pääasiassa käytettiin kirjaston perustoimintoja pienillä lisäyksillä.

Vakuutusten lisääminen kaikille voimassaolovuosille toi mukanaan suurimman testaustoteutuksen, koska ryhmittely oli testattava monien eri ääriarvojen suhteen. Näitä ääriajankohtia edusti vuoden ensimmäinen hetki keskiyön jälkeen sekä vuoden viimeinen hetki ennen keskiyötä. Samoin kirjoitettiin myös testit haastaville alkamis- ja loppumisajoille, esimerkiksi kun kesto oli alle ja yli vuoden, alkamis- ja loppumisajankohdat olivat ääriajankohtia. Lisäksi kokeiltiin niin sanottuja normaaleja alkamis- ja loppumisajankohtia.

Koska HtsyVakuutuksetRestResource-luokalla oli suora riippuvuus kantakyselyjä toteuttavaan VakuuttajaService-luokkaan, käytettiin VakuuttajaService-olion tilalla mock-oliota. Mock-olio palautti testejä palvelevan vastauksen, jolla simuloitiin VakuuttajaService-olion kutsutun metodin vastausta.

Olemassa olevista tuotantolaatuisista ohjelmointikirjastoista käytettyjä luokkia ja metodeja ei testattu, koska on voitava olettaa, että niiden laadunvarmistus on tehty osana kehittäjän kehitys- ja julkaisuprosessia. Sama olettamus tehtiin Java-kielestä, eli ohjelmointikielen itsessään sisältävää toiminnallisuutta, esimerkiksi gettereitä ja settereitä, ei testattu.

Osana insinöörityön toteutusta tehtiin pieniä variaatioita muun muassa olemassa oleviin tietokantakyselyä suorittaviin palveluluokkiin ja päädyttiin katselmoinnin tuloksena, aikataulun ja insinöörityön rajauksen huomioon ottaen, jättämään nämä yksityiskohdat testauksen ulkopuolelle. Aikaisemmin kirjoitettu ohjelmakoodi oli varsinaisen projektin aikaisemmassa vaiheessa jo katselmoitu ja testattu.

Integraatiotestausta sisältävät osuudet, kuten esimerkiksi REST-rajapinnan kutsuminen ja palvelimelta saadun vastauksen testaus, rajattiin pois, koska pääteltiin, että kehitystyöaseman suljetussa testiympäristössä testikannan kanssa komponentit sisältävät liikaa riippuvuuksia palvellakseen tarkoitusta yksikkötesteinä. Nämä asiat toteutetaan lopulta manuaalisessa tai automatisoidussa testauksessa ja käyttöönottotestauksessa. Lisäksi VakuuttajaService-olion käyttäminen yksikkötesteissä olisi muuttanut toteutetun osuuden yksikkötestejä lähemmäksi integraatiotestausta, mikä taas olisi ollut ristiriidassa tavoitteen kanssa. Tavoite oli testata uutta toteutettua ohjelmakoodia.

5 Pohdinta

Tätä projektia ja insinöörityötä tehdessä pohdittiin tulevan insinöörin suoriutumista verrattuna työelämän odotuksiin. Vaikka taustalla oli aina jonkinlaista näkemystä tai ymmärrystä siitä, mitä piti tehdä tai mitä pitää selvittää, että pääsee tekemään, jouduttiin työn etenemisen aikana välillä konsultoimaan kokeneempia kollegoita. Elisa Appelsiini määritteli insinöörityön projektiin laadunvalvonnan työtavoiksi katselmoinnin ja manuaalisen testauksen. Alkuvaiheessa eteneminen oli epävarmempaa, ja toteutusta refaktoroitiin katselmoinnin tulosten perusteella, kunnes suoritus täytti kriteerit. Yksikkötestien tuominen oleelliseksi osaksi insinöörityön laadunvalvontaa ja insinöörityön aihetta toi tukea itsevarmuuteen ja sisäistä laadunvarmistusta. Loppuvaiheessa yksikkötestit nopeuttivat ohjelmointivirheiden selvittelyä ja saivat myös aikaan oma-aloitteista ohjelmakoodin refaktorointia.

Toteutustyön aikana tuli usein käynnistettyä REST-kutsuja suoraan selaimesta tai Postman-ohjelmasta, ja näitä kutsuja tuli rakenneltua Notepad++:ssa. Jälkikäteen tätäkin manuaalista osaa testauksesta olisi pitänyt miettiä automatisoitavaksi: olisiko nämä osuudet voitu automatisoida esimerkiksi skriptikielellä tai muulla tavalla. Postman-sovellus itsessään mahdollistaa REST-API-testien ajamisen ja skriptaamisen, mutta se olisi vaatinut panostusta uusien asioiden opetteluun ja testien tallentamisen Postmanin pilvipalveluun.

Tämä olisi myös lisännyt yhden asian selvitettäväksi tietoturvan näkökulmasta. Tämän projektin aikana puuttuva automaatio ei varsinaisesti laskenut työtehokkuutta, mutta se olisi voinut olla ammattimaisempi ja mielekkäämpi lähestymistapa. Muutamia satunnaisia virheitäkin olisi voitu välttää, kun kopioitiin vääriä testikyselyjä ja ihmeteltiin vääriä tuloksia. Tämä olisi voitu korjata automaatiolla.

Vastaavalla tavalla olisi pitänyt pitää määrätietoisempaa kirjanpitoa hyödyllisistä T-SQL-lausekkeista, joilla sai haettua oleellisia tulosjoukkoja sekä luotua uusia rivejä tietokantaan. Tietokantaoperaatioiden kanssa meni ehkä vähän ylimääräistä aikaa kertauksen ja selvitystyön parissa, vaikka toteutuksessa ei aluksi ollut kyseessä vahvasti tietokantana liittyviä tehtäviä. Toteutuksen aikana huomattiin kuitenkin, että REST-toteutus vaati pieniä muutoksia tietokantakutsuihin, eli pieniä lisäyksiä ja variaatioita olemassa oleviin tietokantaa käyttäviin luokkiin.

Työn aikana kävi myös ilmi, että sovelluskehitystyö ei ole pelkästään ohjelmakoodin tuottamista ja kääntämistä, vaan siihen sisältyi paljon tutkimustyötä, dialogia ja vaihtoehtojen vertailuja. Suureksi avuksi osoittautui muun muassa Googlen hakukone ja Stack Overflow -keskustelupalsta, joista viimeksi mainittu oli suurimpana apuna ongelmien ratkaisussa ja virhekoodien selvittelemisessä.

Projektin aikana nuo pienet muutokset, joita toteutettiin tietokantaa hyödyntäviin palveluluokkiin, veivät jonkin verran aikaa, koska asiaa jouduttiin tutkimaan. Lopullinen toteutus saatiin kuitenkin aikaiseksi tutkinnan ja yritys ja erehdys -metodin avulla. Ohjelmallisesti rakennetut SQL-kyselyt eivät kääntäessä ilmoittaneet ongelmista, mutta ne palauttivat poikkeuksen tietokanta-ajurista ajon aikana. Jos yksikkötestejä olisi toteutettu aikaisemmin tietokantakyselyjä toteuttaviin palveluluokkiin, se olisi auttanut insinöörityöprojektin puitteissa tehtyjen muutosten toteutuksessa.

Kun projektin aikana annettavat tehtävät tulivat tiketteinä tikettijärjestelmään, oli tarkoitus saada yhdellä versionhallinnan vientioperaatiolla tuotua tiketin määrittelemät kehitystyöt. Joissain tilanteissa yhden tiketin kehitystyössä meni enemmän aikaa kuin yksi työpäivä, jolloin vientejä tehtiin useita yhdelle tiketille. Lisäksi vientejä tehtiin katselmointipalautteen tarpeesta. Jälkikäteen ajateltuna olisi myös voinut todeta, että joissain tapauksissa insinöörityöntekijän lähtökohdat huomioon ottaen tikettien tehtävät olisi voitu purkaa vielä muutamaksi tiketiksi, mikä olisi nopeuttanut oikean tavan ja suunnan valintaa.

Koska insinöörityön puitteissa tehtyä toteutusta varten kirjoitettiin yksikkötestit, voitiin todeta, että jotkut ohjelmointivirheet läpäisivät katselmoinnin, mutta jäivät kiinni, kun yksikkötestauksen toteutusta saatiin paremmin käyntiin. Yksikkötesteillä voitiin myös löytää vaikeasti havaittavia vikoja, kuten esimerkiksi se, miten toteutus reagoisi vääränlaisiin syötteisiin tai toimiiko esimerkiksi tietueiden ryhmittely vuosilukujen perusteella. Nimenomaan kyselytulosten järjestelyyn, ryhmittelyyn ja rajaukseen liittyviä potentiaalisia vikoja saatiin kiinni yksikkötestauksen avulla.

Projektissa toteutettua osaa ei vielä viety tuotantoon, vaan jäi vielä nähtäväksi koko laadunvalvonnan ketju eli miten kehitetty komponentti selviää sisäisen laadunvalvonnan manuaalisesta testauksesta ja vastaanottotestauksesta. Insinöörityön kannalta mielenkiintoinen tieto olisi ollut, kuinka paljon ongelmia saatiin suljettua pois jo kehitysvaiheessa valituilla menetelmillä.

Vaikka yksikkötestaus itsessään ei ollut pakollinen osa alkuperäistä toimeksiantoa, se muotoutui merkittäväksi osaksi toteutusta osana insinöörityön ohjausta. Projektin yksikkötestauksesta tuli konkreettista hyötyä Elisa Appelsiinille sekä projektin aikana että sen jälkeen. Yksikkötestaus mahdollisti insinöörityössä itsenäisemmän ja itsevarmemman työskentelyn ja vähensi mahdollisesti katselmointikierroksia ja tuotantokoodiin päätyviä virheitä. Lisäksi tulevaisuudessa, kun joku muu ylläpitää, muuttaa tai tekee toisen version järjestelmän REST-rajapinnasta, on insinöörityöprojektissa toteutettu kokonaisuus dokumentoitu ja varustettu valmiilla yksikkötesteillä. Yksikkötestit voivat helpottaa muutosten tekemistä, etenkin jos kyseessä on joku muu kuin alkuperäinen kehittäjä, kun voidaan nappia painamalla selvittää, tekeekö komponentti vielä saman asian kuin se teki ennen muutosta.

Lähteet

1 Elisa Appelsiini Oy - Keitä olemme? 2014. Verkkodokumentti. Elisa Appelsiini Oy. <http://www.appelsiini.fi/pages/uusi/keitae-olemme.php>. Luettu 20.8.2016.

2 Fielding, Roy Thomas. 2000. Architectural Styles and the Design of Network-based Software Architectures. Verkkodokumentti. University of California, Irvine. <http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf> Luettu 19.12.2015.

3 Richardson, Leonard, Amundsen, Mike & Ruby, Sam. 2013. RESTful Web APIs. Second release. O'Reilly Media.

4 Fielding, et al. 1999. RFC 2161 : Hypertext transfer Protocol – HTTP /1.1. Verkkodokumentti. <https://tools.ietf.org/html/rfc2616>. Luettu 9.8.2016.

5 Berners-Lee, et al. 2005. RFC 3986 : Uniform Resource Identifier (URI) : Generic Syntax. Verkkodokumentti. <https://tools.ietf.org/html/rfc3986>. Luettu 9.8.2016.

6 Gregorio, et al. 2012. RFC 6570 : Uri Template. Verkkodokumentti. <https://tools.ietf.org/html/rfc6570>. Luettu 9.8.2016.

7 Martin, James. 1983. Managing the Data Base Environment. Verkkodokumentti. <https://books.google.fi/books?redir_esc=y&hl=fi&id=ymy4AAAAIAAJ&pg=PA381&sig=ACfU3U1EluSDPcg9pcy_XwJaZXxjZlp3kQ&focus=searchwithinvolume&q=crud>. Luettu 10.8.2016.

8 Create, read, update and delete. 2016. Verkkodokumentti. Wikipedia. <https://en.wikipedia.org/wiki/Create,_read,_update_and_delete>. Luettu 9.8.2016.

9 Bray. 2014. RFC 7159. The JavaScript Object Notation (JSON) Data Interchange Format. Verkkodokumentti. <https://tools.ietf.org/html/rfc7159> . Luettu 10.8.2016.

10 Sperberg-Queen C.M.& Thompson Henry. 2004. XML Schema. Verkkodokumentti. <https://www.w3.org/XML/Schema>. Luettu 5.9.2016.

11 Kawaguchi, K., Vajjhala, S.& Fialli, J. 2009. JSR-000222 The Java Architecture for XML Binding (JAXB) 2.2. Verkkodokumentti. <http://download.oracle.com/otndocs/jcp/jaxb-2.2-mrel2a-oth-JSpec/>. Luettu 5.9.2016.

12 Galiegue, et al. 2013. JSON Schema: core definitions and terminology draft-zyp-json-schema-04. Verkkodokumentti. <https://tools.ietf.org/id/draft-zyp-json-schema-04.txt>. Luettu 5.9.2016.

13 Pericas-Geertsen, S.& Potociar, M. 2014. JSR-000339 Java API for RESTful Web Services 2.0 Final Release for Evaluation. Verkkodokumentti. <http://download.oracle.com/otndocs/jcp/jaxrs-2_0-fr-eval-spec/index.html>. Luettu 23.8.2016.

14 DeMichiel, L.& Shannon, B. 2013. JSR-000342 Java Platform, Enterprise Edition (Java EE) Specification, v7. Verkkodokumentti. <http://download.oracle.com/otndocs/jcp/java_ee-7-fr-eval-spec/index.html>. Luettu 23.8.2016.

15 Acharya, Sujoy. 2014. Mastering Unit Testing Using Mockito and JUnit. Packt Publishing. Kindle e-kirja, Amazon.com.

16 Osherove, Roy. 2014. The Art of Unit Testing. Second Edition: with examples in C#. E-kirja. Manning Publications.

17 Black Box Testing. 2016. Verkkodokumentti. Software Testing Fundamentals. <http://softwaretestingfundamentals.com/black-box-testing/>. Luettu 12.8.2016.

18 White Box Testing. 2016. Verkkodokumentti. Software Testing Fundamentals. <http://softwaretestingfundamentals.com/white-box-testing/>. Luettu 12.8.2016.

19 Gray Box Testing. 2016. Verkkodokumentti. Software Testing Fundamentals. <http://softwaretestingfundamentals.com/gray-box-testing/>. Luettu 12.8.2016.

20 JUnit – About. 2016. Verkkodokumentti. junit.org. <http://junit.org/junit4>. Luettu 16.8.2016.

21 Clark, Mike. 2016. JUnit – Frequently Asked Questions. Verkkodokumentti. <http://junit.org/junit4/faq.html>. Luettu 1.11.2016.

22 Allen, et al. Arquillian: An integration testing framework for Java EE. Verkkodokumentti <https://docs.jboss.org/arquillian/reference/1.0.0.Alpha1/en-US/html_single/>. Luettu 21.10.2016.

23 Macke, Stefan. 2016. How to seed the database with sample data for an Arquillian test. Verkkodokumentti. <http://serviceorientedarchitect.com/how-to-seed-the-database-with-sample-data-for-an-arquillian-test/>. Luettu 21.10.2016.

src/main/java/model/Entities.java

```
package model;

import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

/**
 * Entities.java
 * <p>
 * Class for containing a collection of objects of class Entity.
 * <p>
 * Example written as part of bachelor of engineering thesis.
 * @author Erkki Keränen
*/
@XmlRootElement (name = "Entities")
public class Entities {

		/**
		 * The list that contains the elements as objects of type Entity.
		 */
		@XmlElement(name = "Entity")
		public List<Entity> entities;

		public Entities()
		{
			entities = new ArrayList<Entity>();
		}

}
```

src/main/java/model/Entity.java

```
package model;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

/** Entity.java
 * <p>
 * Class for containing an entity.
 * <p>
 * Example written as part of bachelor of engineering thesis.
 * @author Erkki Keränen
*/
@XmlRootElement(name = "Entity")
public class Entity {

		/**
		 * The id of the entity.
		 */
		private int id;

		/**
		 * The name (a field) of the entity
		 */
		private String name;

		@XmlElement
		public int getId() {
				return id;
		}
		public void setId(int id) {
				this.id = id;
		}

		@XmlElement
		public String getName() {
				return name;
		}
		public void setName(String name) {
				this.name = name;
		}

}
```

README.md – Esimerkkiprojektin kuvaus ja ohjeet

````
### Simple Jax-RS REST-Api with unit tests and integration tests

https://gitlab.com/erkkikeranen/rest-api-with-junit-and-arquillian

This repository contains the example code for my bachelor of engineering in software
engineering degree project. This projects aim is not to be fully correct
in all aspects. Instead it tries to demonstrate the difference on writing unit tests with
mocks and integration tests, and what kind of results these two
types/phases of testing can deliver.


* Simple CRUD mapping to in memory repository
* JUnit unit tests with Mockito and Arquillian integration tests for comparision


Remember to set $JBOSS_HOME:
```
export JBOSS_HOME=/opt/wildfly-10.1.0-Final/
```

(or whatever is your path to wildfly, if local instance)

To run unit tests (Junit & Mockito) on local/remote wildfly instance:

```
mvn test -Punit-test
```

To run all tests (Junit & Mockito and Arquillian) on local/remote wildfly instance:

```
mvn integration-test -Parquillian-wildfly-remote
```

To deploy on wildfly
```
mvn wildfly:deploy
```
The deployment should appear on http://localhost:8080/restapiexample/rest/api (or whatever
your configuration looks alike)

---
## Sources of information

* Using Forge, existing Java EE Project was configured for Arquillian with help from
http://blog.arungupta.me/enable-arquillian-existing-javaee-project-techtip54/
*   Separating   tests   in   Maven   with   help   from   https://thomassundberg.word-
press.com/2012/08/21/separating-tests-in-maven/
````

pom.xml – Esimerkkiprojektin projektimalli

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
             http://maven.apache.org/xsd/maven-4.0.0.xsd"
             xmlns="http://maven.apache.org/POM/4.0.0"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <modelVersion>4.0.0</modelVersion>
  <groupId>erkkikeranen</groupId>
  <artifactId>restapiexample</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>war</packaging>
  <name>REST API example</name>
  <description>REST API example with Unit tests, mocking and Arquillian integration tests.
Based on jboss-javaee6-webapp archetype</description>
  <url>https://gitlab.com/erkkikeranen/rest-api-with-junit-and-arquillian</url>

  <licenses>
    <license>
      <name>The MIT License (MIT)</name>
      <url>https://opensource.org/licenses/MIT</url>
      <distribution>manual</distribution>
    </license>
  </licenses>

  <developers>
    <developer>
              <id>erkkikeranen</id>
              <name>Erkki Keränen</name>
    </developer>
  </developers>

  <properties>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <version.compiler.plugin>3.1</version.compiler.plugin>
    <version.jboss.bom>8.2.1.Final</version.jboss.bom>
    <version.surefire.plugin>2.16</version.surefire.plugin>
    <version.war.plugin>2.5</version.war.plugin>
    <version.wildfly.maven.plugin>1.0.2.Final</version.wildfly.maven.plugin>
  </properties>

  <!--  Define standard versions for dependencies -->
  <dependencyManagement>
    <dependencies>

              <!--  WildFly JBoss Java EE 7 Specification APIs With Tools
                          Dependency Management for Java EE 7 Specification APIs with
                          Deployment and Testing Tools -->
      <dependency>
        <groupId>org.wildfly.bom</groupId>
        <artifactId>jboss-javaee-7.0-with-tools</artifactId>
        <version>${version.jboss.bom}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>

              <!-- WildFly JBoss Java EE 7 Specification APIs With Hibernate 3
                         Dependency Management for Java EE 7 Specification APIs with
Hibernate 3 projects -->
      <dependency>
        <groupId>org.wildfly.bom</groupId>
        <artifactId>jboss-javaee-7.0-with-hibernate</artifactId>
        <version>${version.jboss.bom}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
```

```
            <!-- Arquillian BOM
         Arquillian Bill Of Material -->
      <dependency>
        <groupId>org.jboss.arquillian</groupId>
        <artifactId>arquillian-bom</artifactId>
        <version>1.1.11.Final</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>

  </dependencies>
</dependencyManagement>

<!-- Project dependencies -->
<dependencies>

           <!-- APIs for CDI (Contexts and Dependency Injection for Java)
                        Provided by dependency management -->
  <dependency>
    <groupId>javax.enterprise</groupId>
    <artifactId>cdi-api</artifactId>
    <scope>provided</scope>
  </dependency>

           <!-- Common Annotations API
                        JSR 250: Common Annotations for the Java(TM) Platform
                        Provided by dependency management -->
  <dependency>
    <groupId>org.jboss.spec.javax.annotation</groupId>
    <artifactId>jboss-annotations-api_1.2_spec</artifactId>
    <scope>provided</scope>
  </dependency>

           <!-- JAX RS Core API 2.0
               Provided by dependency management -->
  <dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>jaxrs-api</artifactId>
    <scope>provided</scope>
  </dependency>

           <!--  RESTEasy JAXB Provider
                RESTEasy JAX-RS support for (un)marshalling JAXB annotated classes -->
  <dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jaxb-provider</artifactId>
    <version>3.0.16.Final</version>
  </dependency>

           <!-- JUnit unit testing framework -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <scope>test</scope>
  </dependency>

  <!--  Mockito mocking framework for JUnit -->
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.10.19</version>
    <scope>test</scope>
           </dependency>

           <!-- Arquillian TestRunner JUnit Container
                        JUnit Container Implementation for the Arquillian Project -->
  <dependency>
    <groupId>org.jboss.arquillian.junit</groupId>
    <artifactId>arquillian-junit-container</artifactId>
    <scope>test</scope>
  </dependency>

  <!-- Arquillian Protocol Servlet
      Protocol handler for communicating using a
```

```
        servlet / http following the Servlet 2.5/ 2.5/.x spec. -->
    <dependency>
      <groupId>org.jboss.arquillian.protocol</groupId>
      <artifactId>arquillian-protocol-servlet</artifactId>
      <scope>test</scope>
    </dependency>

    <!--  RestEasy fix
              RESTEasy JAX RS Client-->
    <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-client</artifactId>
      <version>3.0.5.Final</version>
      <scope>test</scope>
    </dependency>

              <!-- https://mvnrepository.com/artifact/org.jboss.resteasy/resteasy-jaxrs
              In case : Import to fix java.lang.RuntimeException: java.lang.ClassNot-
FoundException:
              org.jboss.resteasy.spi.ResteasyProviderFactory -->

              <!--  Commented Out
              <dependency>
      <groupId>org.jboss.resteasy</groupId>
                <artifactId>resteasy-jaxrs</artifactId>
      <version>3.0.19.Final</version>
              </dependency>
              Commented Out -->

  </dependencies>

  <build>
    <!-- The name of the deployment -->
    <finalName>${project.artifactId}</finalName>
    <plugins>

      <!-- Maven WAR packager -->
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>${version.war.plugin}</version>
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>

      <!-- Plugin for controlling wildfly with maven -->
      <plugin>
        <groupId>org.wildfly.plugins</groupId>
        <artifactId>wildfly-maven-plugin</artifactId>
        <version>${version.wildfly.maven.plugin}</version>
      </plugin>

      <!-- Maven Surefire plugin runs tests and creates Reports.
          run unit tests with mvn test -Ptest,
          Arquillian tests with mvn integration-test -Parquillian-wildfly-remote -->

      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>${version.surefire.plugin}</version>
         <executions>

             <execution>
                 <id>default-test</id>
                 <configuration>
                     <includes>
                         <include>**/*JUnitTest.java</include>
                     </includes>
                     <excludes>
                         <include>**/*ArquillianTest.java</include>
                     </excludes>
                 </configuration>
             </execution>
```

```
            <execution>
                <id>integration-test</id>
                <phase>integration-test</phase>
                <goals>
                    <goal>test</goal>
                </goals>
                <configuration>
                    <includes>
                        <include>**/*ArquillianTest.java</include>
                    </includes>
                    <excludes>
                        <include>**/*JUnitTest.java</include>
                    </excludes>
                </configuration>
            </execution>

        </executions>
    </plugin>

  </plugins>
</build>

<!--  Build profiles -->
<profiles>

  <!-- default profile
            Skip tests-->
  <profile>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>${version.surefire.plugin}</version>
          <configuration>
            <skip>true</skip>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>

  <!-- unit-test profile
            run unit tests -->
  <profile>
    <id>unit-test</id>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>${version.surefire.plugin}</version>
          <configuration>
            <skip>false</skip>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>

  <profile>
    <id>arq-wildfly-managed</id>
    <dependencies>
      <dependency>
        <groupId>org.wildfly</groupId>
        <artifactId>wildfly-arquillian-container-managed</artifactId>
        <scope>test</scope>
      </dependency>
    </dependencies>
  </profile>

  <!-- arquillian-wildfly-remote profile
      for starting up, deploying tests, running tests,
```

```
      undeploying tests and shutting down wildfly container -->
  <profile>
    <id>arquillian-wildfly-remote</id>
    <dependencies>
      <dependency>
        <groupId>org.wildfly</groupId>
        <artifactId>wildfly-arquillian-container-managed</artifactId>
        <scope>test</scope>
                        </dependency>
    </dependencies>
  </profile>

  <profile>
    <id>openshift</id>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-war-plugin</artifactId>
          <version>${version.war.plugin}</version>
          <configuration>
            <outputDirectory>deployments</outputDirectory>
            <warName>ROOT</warName>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>

  <profile>
    <id>arq-wildfly-remote</id>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>2.14.1</version>
          <configuration>
            <systemPropertyVariables>
              <arquillian.launch>arquillian-wildfly-remote</arquillian.launch>
            </systemPropertyVariables>
          </configuration>
        </plugin>
      </plugins>
    </build>
    <dependencies>
      <dependency>
        <groupId>org.wildfly</groupId>
        <artifactId>wildfly-arquillian-container-remote</artifactId>
        <version>8.2.1.Final</version>
        <scope>test</scope>
      </dependency>
    </dependencies>
  </profile>

</profiles>
</project>
```

src/main/java/rest/JaxRsActivator.java

```
package rest;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

/**
 * JaxRsActivator.java
 * <p>
 * RESTEasy "no XML" approach for activating JAX-RS.
 * <p>
 * Example written as part of bachelor of engineering thesis.
 * @author Erkki Keränen
 *
*/
@ApplicationPath("/rest")
public class JaxRsActivator extends Application{
              /* Intentionally left blank */
}
```

src/main/java/rest/RestService.java

```
/* RestService.java */

package rest;

import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.DELETE;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import data.DataRepository;
import model.Entity;
/**
 * RestService.java
 * <p>
 * Simple JAX-RS Rest Service providing CRUD operations to an in-memory data repository.
 * <p>
 * Example written as part of bachelor of engineering thesis.
 * @author Erkki Keränen
*/
@Path("/api")
@RequestScoped
public class RestService {

		/**
		 * The DataRepository that is injected for every service request.
		 */
		@Inject
		DataRepository dataRepository;

		/**
		 * Dependency injector for mock-objects and Arquillian tests
		 * @param dataRepository For testing instantiated DataRepository object or
mock.
		 */
		public void setDataRepository(DataRepository dataRepository) {
				this.dataRepository = dataRepository;
		}

		/**
		 * HTTP POST method providing mapping to DataRepository CRUD Create.
		 * <p>
		 * For the simplicity of this example, this method always succeeds.
		 * @param entity The entity to add to repository.
		 * @return HTTP Response with HTTP Status 201 Created.
		 */
		@POST
		@Consumes({MediaType.APPLICATION_JSON,
							MediaType.APPLICATION_XML})
		public Response create(Entity entity)
		{
				dataRepository.create(entity);
				return Response.created(null).build();
		}

		/**
		 * HTTP GET method providing mapping to DataRepository CRUD Read (All)
		 * @return HTPP Response with HTTP Status 200 Ok with body containing all
entities from the DataRepository.
		 */
		@GET
```

```
		@Produces({MediaType.APPLICATION_JSON,
                                                MediaType.APPLICATION_XML})
		public Response readAll()
		{
			return Response.
                                                ok().
                                                entity(dataRepository.readAll()).
                                                build();
		}

		/**
		 * HTTP GET method providing mapping to DataRepository CRUD Read
		 * @param id The id of the entity to read.
		 * @return On success, HTPP Response with HTTP Status 200 Ok with body containing the entity.
		 * If fail, HTTP Response with HTTP Status 404 Not Found.
		 */
		@GET
		@Path("/{id}")
		@Produces({MediaType.APPLICATION_JSON,
                                                MediaType.APPLICATION_XML})
		public Response read(@PathParam("id") int id)
		{
			Entity entity = dataRepository.read(id);
			if (entity == null)
				return Response
                                                                .status(404).build();

			return Response
                                                .ok().
                                                entity(entity).
                                                build();
		}
		/**
		 * HTTP PUT method providing mapping to DataRepository CRUD Update
		 * @param entity The new contents containing the actual id, which defines the entity for the operation.
		 * @return On success, HTTP Status 200 Ok. If fail, HTTP Status 400 Bad Request.
		 */
		@PUT
		@Consumes({MediaType.APPLICATION_JSON,
                                                MediaType.APPLICATION_XML})
		public Response update(Entity entity)
		{
			if (dataRepository.update(entity))
				return Response.ok().build();

			return Response.status(400).build();
		}

		/**
		 * HTTP DELETE method providing mapping to DataRepository CRUD Delete
		 * @param id The id of the entity to delete.
		 * @return On success, HTTP Status 204 No Content. If fail, HTTP Status 404 Not Found.
		 */
		@DELETE
		@Path("/{id}")
		@Produces({MediaType.APPLICATION_JSON,
                                                MediaType.APPLICATION_XML})
		public Response delete(@PathParam("id") int id)
		{
			if (dataRepository.delete(id))
				return Response.noContent().build();

			return Response.status(404).build();
		}
}
```

src/main/java/data/DataRepository.java

```
package data;

import javax.enterprise.context.ApplicationScoped;
import model.Entity;
import model.Entities;

/**
 * DataRepository.java
 * <p>
 * A simple in-memory repository for storing and manipulating data with CRUD operations.
 * <p>
 * Example written as part of bachelor of engineering thesis.
 * @author Erkki Keränen
*/
@ApplicationScoped
public class DataRepository {

		/**
		 * The next available id for the data, that is used for
		 * populating the id field of an entity.
		 */
		private int nextId;

		/**
		 * The object containing the data used by the DataRepository.
		 */
		private Entities data;

		public DataRepository()
		{
			data = new Entities();
			nextId = 0;
		}

		/**
		 * DataRepository CRUD Create
		 * @param entity Entity to add in the DataRepository.
		 */
		public void create(Entity entity)
		{
			entity.setId(nextId);
			data.entities.add(entity);
			nextId++;
		}

		/**
		 * DataRepository CRUD Read (All)
		 * @return All entities.
		 */
		public Entities readAll()
		{
			return data;
		}

		/**
		 * DataRepository CRUD Read
		 * @param id The entity with selected id to retrieve.
		 * @return The entity with given id, or null if not found.
		 */
		public Entity read(int id)
		{
			Entity result = null;

			for (Entity entity : data.entities)
			{
				if (entity.getId() == id)
				{
					result = entity;
```

```
                                                  break;
                                        }
                              }
                              return result;
                    }

                    /**
                     * DataRepository CRUD Update
                     * @param entity Entity containing valid id with new values.
                     * @return True if success, false if fail.
                     */
                    public boolean update(Entity entity)
                    {
                              for (int i=0;i<data.entities.size();i++)
                              {
                                        if    (data.entities.get(i).getId()    ==    en-
tity.getId())
                                        {
                                                  data.entities.get(i).setName(en-
tity.getName());
                                                  return true;
                                        }
                              }
                              return false;
                    }

                    /**
                     * DataRepository CRUD Delete
                     * @param id The id of the given entity to delete.
                     * @return True if success, false if fail.
                     */
                    public boolean delete(int id)
                    {
                              for (int i=0;i<data.entities.size();i++)
                              {
                                        if (data.entities.get(i).getId() == id)
                                        {
                                                  data.entities.remove(i);
                                                  return true;
                                        }
                              }
                              return false;
                    }
}
```

src/test/java/engineering/DataRepositoryJUnitTest.java

```
package engineering;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

import data.DataRepository;
import model.Entity;

/**
 * DataRepositoryJunitTest.java
 * <p>
 * Demonstrates unit testing of the in-memory Data Repository.
 * @author Erkki Keränen
 *
 */
public class DataRepositoryJUnitTest {

		DataRepository testDataRepository;
		Entity testEntity1;
		Entity testEntity2;

		final static int TEST_ENTITY1_ID = 123;
		final static String TEST_ENTITY1_NAME = "Test Name1";


		final static int TEST_ENTITY2_ID = 456;
		final static String TEST_ENTITY2_NAME = "Test Name2";

		final static String TEST_ENTITY3_NAME = "Modified Name";

		@BeforeClass
		public static void setUpBeforeClass() throws Exception {
				/* Time and resource expensive setup
				 * before all test runs of
				 * this class should be put here
				 */
		}

		@AfterClass
		public static void tearDownAfterClass() throws Exception {
				/* Time and resource expensive tear
				 * down after all test runs of
				 * this class should be put here
				 */
		}

		@Before
		public void setUp() throws Exception {
				/* We reconstruct the datarepository and test entities before
all runs */
				testDataRepository = new DataRepository();

				testEntity1 = new Entity();
				testEntity1.setId(TEST_ENTITY1_ID);
				testEntity1.setName(TEST_ENTITY1_NAME);

				testEntity2 = new Entity();
				testEntity2.setId(TEST_ENTITY2_ID);
				testEntity2.setName(TEST_ENTITY2_NAME);
		}

		@After
```

```
	public void tearDown() throws Exception {
			/* We empty the datarepository and test entities after all runs
*/
			testDataRepository = null;
			testEntity1 = null;
			testEntity2 = null;
	}

	@Test
	public void testDataRepositoryConstructorConstructsEmptyContainerNotNull()
	{
			assertNotNull("DataRepository was not constructed correctly",
testDataRepository.readAll());
	}

	@Test
	public void testDataRepositoryEmptyAfterConstruction()
	{
			assertEquals("The repository was not empty after construc-
tion",
					0,
					testDataRepository.readAll().enti-
ties.size());
	}

	@Test
	public void testDataRepositoryCreateAndVerifyAmountOfEntities() {

			assertEquals("The repository was not empty after construc-
tion",
					0,
					testDataRepository.readAll().enti-
ties.size());

			testDataRepository.create(testEntity1);
			assertEquals("Wrong amount of entities in repository",
					1,
					testDataRepository.readAll().enti-
ties.size());

			testDataRepository.create(testEntity2);
			assertEquals("Wrong amount of entities in repository",
					2,
					testDataRepository.readAll().enti-
ties.size());

	}

	@Test
	public void testCheckCreatedDataWasCorrect() {

			testDataRepository.create(testEntity1);
			testDataRepository.create(testEntity2);

			assertEquals("There was wrong amount of entities in the repos-
itory",
					2,
					testDataRepository.readAll().enti-
ties.size());

			assertEquals("The entity contained incorrect data",
					TEST_ENTITY1_NAME,
					testDataRepository.readAll().enti-
ties.get(0).getName());

			assertEquals("The entity contained incorrect data",
					TEST_ENTITY2_NAME,
					testDataRepository.readAll().enti-
ties.get(1).getName());

	}

	@Test
	public void testMostRecentCreatedEntitiesCorrect() {
```

```
                testDataRepository.create(testEntity1);

                assertEquals("Most recent entity did not equal the last cre-
ated",
                                                TEST_ENTITY1_NAME,
                                                testDataRepository.read(testData-
Repository.readAll().entities.size()-1).getName());

                testDataRepository.create(testEntity2);

                assertEquals("Most recent entity did not equal the last cre-
ated",
                                                TEST_ENTITY2_NAME,
                                                testDataRepository.read(testData-
Repository.readAll().entities.size()-1).getName());
        }

        @Test
        public void testGetNullWhenReadingFirstEntityFromEmptyRepository() {

                assertNull("The requested entity that was not found did not
result as null",
                                                testDataRepository.read(0));
        }

        @Test(expected=NullPointerException.class)
        public void testThrowNullPointerExceptionWhenUsingGetNameOnNullEntityInEmp-
tyRepository() {
                @SuppressWarnings("unused")
                String testTheNameReadFromEmptyEntityInEmptyRepository = test-
DataRepository.read(0).getName();
        }

        @Test
        public void testUpdateEntity1() {

                testDataRepository.create(testEntity1);
                testDataRepository.create(testEntity2);

                assertEquals("The requested entity was not the same as ex-
pected",
                                                TEST_ENTITY1_NAME,
                                                testDataRepository.read(0).get-
Name());
                assertEquals("The requested entity was not the same as ex-
pected",
                                                TEST_ENTITY2_NAME,
                                                testDataRepository.read(1).get-
Name());

                Entity testEntityForUpdatingData = new Entity();

                testEntityForUpdatingData.setId(0);
                testEntityForUpdatingData.setName(TEST_ENTITY3_NAME);

                assertTrue("Update should succeed",
                                                testDataRepository.up-
date(testEntityForUpdatingData));
                assertEquals("The requested entity was not changed",
                                                TEST_ENTITY3_NAME,
                                                testDataRepository.read(0).get-
Name());
                assertEquals("Other entity than requested was changed",
                                                TEST_ENTITY2_NAME,
                                                testDataRepository.read(1).get-
Name());
        }

        @Test
        public void testUpdateEntity2() {

                testDataRepository.create(testEntity1);
                testDataRepository.create(testEntity2);
```

```
                        assertEquals("The requested entity was not the same as ex-
pected",
                                                TEST_ENTITY1_NAME,
                                                testDataRepository.read(0).get-
Name());
                        assertEquals("The requested entity was not the same as ex-
pected",
                                                TEST_ENTITY2_NAME,
                                                testDataRepository.read(1).get-
Name());

                        Entity testEntityForUpdatingData = new Entity();

                        testEntityForUpdatingData.setId(1);
                        testEntityForUpdatingData.setName(TEST_ENTITY3_NAME);

                        assertTrue("Update should succeed",
                                                testDataRepository.up-
date(testEntityForUpdatingData));
                        assertEquals("The requested entity was not changed",
                                                TEST_ENTITY3_NAME,
                                                testDataRepository.read(1).get-
Name());
                        assertEquals("Other entity than requested was changed",
                                                TEST_ENTITY1_NAME,
                                                testDataRepository.read(0).get-
Name());
                }

                @Test
                public void testUpdateEntityThatDoesNotExist1() {
                        Entity testEntityForUpdatingData = new Entity();

                        testEntityForUpdatingData.setId(0);
                        testEntityForUpdatingData.setName(TEST_ENTITY3_NAME);

                        assertEquals("There was wrong amount of entities in the repos-
itory",
                                                0,
                                                testDataRepository.readAll().enti-
ties.size());

                        assertFalse("Update should not have succeeded on non-existing
entity",
                                                testDataRepository.up-
date(testEntityForUpdatingData));


                }

                @Test
                public void testUpdateEntityThatDoesNotExist2() {

                        testDataRepository.create(testEntity1);
                        Entity testEntityForUpdatingData = new Entity();

                        testEntityForUpdatingData.setId(1);
                        testEntityForUpdatingData.setName(TEST_ENTITY3_NAME);

                        assertEquals("There was wrong amount of entities in the repos-
itory",
                                                1,
                                                testDataRepository.readAll().enti-
ties.size());

                        assertFalse("Update should not have succeeded on non-existing
entity",
                                                testDataRepository.up-
date(testEntityForUpdatingData));


                }
```

```
	@Test
	public void testUpdateEntityThatDoesNotExist3() {

			testDataRepository.create(testEntity1);
			testDataRepository.delete(0);
			testDataRepository.create(testEntity2);
			Entity testEntityForUpdatingData = new Entity();

			testEntityForUpdatingData.setId(0);
			testEntityForUpdatingData.setName(TEST_ENTITY3_NAME);

			assertEquals("There was wrong amount of entities in the repos-
itory",
							1,
							testDataRepository.readAll().enti-
ties.size());

			assertFalse("Update should not have succeeded on non-existing
entity",
							testDataRepository.up-
date(testEntityForUpdatingData));

	}

	@Test
	public void testDeleteObjectThatDoesNotExist() {

			assertFalse("There may not exist an object with the requested
id",
							testDataRepository.delete(0));
	}

	@Test
	public void testDeleteEntity() {

			testDataRepository.create(testEntity1);

			assertTrue("Entity could not be removed from repository with
requested id",
							testDataRepository.delete(0));
	}

	@Test
	public void testDeleteEntititiesFirstInFirstOutOrder() {

			testDataRepository.create(testEntity1);
			testDataRepository.create(testEntity2);

			assertTrue("Entity could not be removed from repository with
requested id",
							testDataRepository.delete(0));

			assertTrue("Entity could not be removed from repository with
requested id",
							testDataRepository.delete(1));
	}

	@Test
	public void testDeleteEntitiesLastInFirstOutOrder() {

			testDataRepository.create(testEntity1);
			testDataRepository.create(testEntity2);

			assertTrue("Entity could not be removed from repository with
requested id",
							testDataRepository.delete(1));

			assertTrue("Entity could not be removed from repository with
requested id",
							testDataRepository.delete(0));
	}

	@Test
	public void testCheckIdNumberingAfterCreatingSequentialEntities() {
```

```
                        for (int i=0;i<5;i++)
                        {
                                testDataRepository.create(testEntity1);
                                assertEquals("Id's were not created in sequential
order",
                                                                        i,
                                                                        testDataRepo-
sitory.read(i).getId());
                        }
                }

                @Test
                public void testCheckIdNumberingAfterCreatingSequnceAndDeletingSequence() {

                        for (int i=0;i<5;i++)
                        {
                                testDataRepository.create(testEntity1);
                                testDataRepository.delete(i);
                        }

                        testDataRepository.create(testEntity1);

                        assertEquals("Wrong amount of entities in repository",
                                                        1,
                                                        testDataRepository.readAll().enti-
ties.size());
                        assertEquals("Next assigned id was not correct",
                                                        5,
                                                        testDataRepository.readAll().enti-
ties.get(0).getId());
                }

                /*
                 * This test fails because the problem
                 * is not addressed in DataRepository
                 * implementation.
                 * This demonstrates the problems
                 * mock objects cannot address,
                 * but can be caught in integration
                 * test.
                 * Hence test ignored.
                 */
                @Ignore
                @Test
                public void testCreateNullObject()
                {
                        testDataRepository.create(null);
                        // Assertions missing
                }
}
```

src/test/resources/arquillian.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<arquillian                                    xmlns="http://jboss.org/schema/arquillian"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"                      xsi:schemaLoca-
tion="http://jboss.org/schema/arquillian          http://jboss.org/schema/arquillian/ar-
quillian_1_0.xsd">
  <container qualifier="arquillian-wildfly-remote" default="true"/>
</arquillian>
```

src/test/java/engineering/DataRepositoryArquillianTest.java

```
package engineering;

import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.jboss.shrinkwrap.api.spec.JavaArchive;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;
import org.junit.runner.RunWith;

import data.DataRepository;
import model.Entity;

/**
 * DataRepositoryArquillianTest.java
 * <p>
 * Demonstrates unit testing of the in-memory Data Repository.
 * Same as DataReposiroryJUnitTest.java, but runs with Arquillian test runner.
 * @author Erkki Keränen
 *
 */
@RunWith(Arquillian.class)
public class DataRepositoryArquillianTest {


		DataRepository testDataRepository;
		Entity testEntity1;
		Entity testEntity2;

		final static int TEST_ENTITY1_ID = 123;
		final static String TEST_ENTITY1_NAME = "Test Name1";


		final static int TEST_ENTITY2_ID = 456;
		final static String TEST_ENTITY2_NAME = "Test Name2";

		final static String TEST_ENTITY3_NAME = "Modified Name";

		@BeforeClass
		public static void setUpBeforeClass() throws Exception {
			/* Time and resource expensive setup
			 * before all test runs of
			 * this class should be put here
			 */
		}

		@AfterClass
		public static void tearDownAfterClass() throws Exception {
			/* Time and resource expensive tear
			 * down after all test runs of
			 * this class should be put here
			 */
		}

		@Before
		public void setUp() throws Exception {
			/* We reconstruct the datarepository and test entities before
all runs */
			testDataRepository = new DataRepository();
```

```
                        testEntity1 = new Entity();
                        testEntity1.setId(TEST_ENTITY1_ID);
                        testEntity1.setName(TEST_ENTITY1_NAME);

                        testEntity2 = new Entity();
                        testEntity2.setId(TEST_ENTITY2_ID);
                        testEntity2.setName(TEST_ENTITY2_NAME);
        }

        @After
        public void tearDown() throws Exception {
                        /* We empty the datarepository and test entities after all runs
*/
                        testDataRepository = null;
                        testEntity1 = null;
                        testEntity2 = null;
        }

        @Test
        public void testDataRepositoryConstructorConstructsEmptyContainerNotNull()
        {
                        assertNotNull("DataRepository was not constructed correctly",
testDataRepository.readAll());
        }

        @Test
        public void testDataRepositoryEmptyAfterConstruction()
        {
                        assertEquals("The repository was not empty after construc-
tion",
                                                0,
                                                testDataRepository.readAll().enti-
ties.size());
        }

        @Test
        public void testDataRepositoryCreateAndVerifyAmountOfEntities() {

                        assertEquals("The repository was not empty after construc-
tion",
                                                0,
                                                testDataRepository.readAll().enti-
ties.size());

                        testDataRepository.create(testEntity1);
                        assertEquals("Wrong amount of entities in repository",
                                                1,
                                                testDataRepository.readAll().enti-
ties.size());

                        testDataRepository.create(testEntity2);
                        assertEquals("Wrong amount of entities in repository",
                                                2,
                                                testDataRepository.readAll().enti-
ties.size());

        }

        @Test
        public void testCheckCreatedDataWasCorrect() {

                        testDataRepository.create(testEntity1);
                        testDataRepository.create(testEntity2);

                        assertEquals("There was wrong amount of entities in the repos-
itory",
                                                2,
                                                testDataRepository.readAll().enti-
ties.size());

                        assertEquals("The entity contained incorrect data",
                                                TEST_ENTITY1_NAME,
                                                testDataRepository.readAll().enti-
ties.get(0).getName());
```

```
                        assertEquals("The entity contained incorrect data",
                                                TEST_ENTITY2_NAME,
                                                testDataRepository.readAll().enti-
ties.get(1).getName());

            }

            @Test
            public void testMostRecentCreatedEntitiesCorrect() {

                        testDataRepository.create(testEntity1);

                        assertEquals("Most recent entity did not equal the last cre-
ated",
                                                TEST_ENTITY1_NAME,
                                                testDataRepository.read(testData-
Repository.readAll().entities.size()-1).getName());

                        testDataRepository.create(testEntity2);

                        assertEquals("Most recent entity did not equal the last cre-
ated",
                                                TEST_ENTITY2_NAME,
                                                testDataRepository.read(testData-
Repository.readAll().entities.size()-1).getName());
            }

            @Test
            public void testGetNullWhenReadingFirstEntityFromEmptyRepository() {

                        assertNull("The requested entity that was not found did not
result as null",
                                                testDataRepository.read(0));
            }

            @Test(expected=NullPointerException.class)
            public void testThrowNullPointerExceptionWhenUsingGetNameOnNullEntityInEmp-
tyRepository() {
                        @SuppressWarnings("unused")
                        String testTheNameReadFromEmptyEntityInEmptyRepository = test-
DataRepository.read(0).getName();
            }

            @Test
            public void testUpdateEntity1() {

                        testDataRepository.create(testEntity1);
                        testDataRepository.create(testEntity2);

                        assertEquals("The requested entity was not the same as ex-
pected",
                                                TEST_ENTITY1_NAME,
                                                testDataRepository.read(0).get-
Name());
                        assertEquals("The requested entity was not the same as ex-
pected",
                                                TEST_ENTITY2_NAME,
                                                testDataRepository.read(1).get-
Name());

                        Entity testEntityForUpdatingData = new Entity();

                        testEntityForUpdatingData.setId(0);
                        testEntityForUpdatingData.setName(TEST_ENTITY3_NAME);

                        assertTrue("Update should succeed",
                                                testDataRepository.up-
date(testEntityForUpdatingData));
                        assertEquals("The requested entity was not changed",
                                                TEST_ENTITY3_NAME,
                                                testDataRepository.read(0).get-
Name());
                        assertEquals("Other entity than requested was changed",
```

```
                                                TEST_ENTITY2_NAME,
                                                testDataRepository.read(1).get-
Name());
        }

        @Test
        public void testUpdateEntity2() {

                testDataRepository.create(testEntity1);
                testDataRepository.create(testEntity2);

                assertEquals("The requested entity was not the same as ex-
pected",
                                                TEST_ENTITY1_NAME,
                                                testDataRepository.read(0).get-
Name());
                assertEquals("The requested entity was not the same as ex-
pected",
                                                TEST_ENTITY2_NAME,
                                                testDataRepository.read(1).get-
Name());

                Entity testEntityForUpdatingData = new Entity();

                testEntityForUpdatingData.setId(1);
                testEntityForUpdatingData.setName(TEST_ENTITY3_NAME);

                assertTrue("Update should succeed",
                                                testDataRepository.up-
date(testEntityForUpdatingData));
                assertEquals("The requested entity was not changed",
                                                TEST_ENTITY3_NAME,
                                                testDataRepository.read(1).get-
Name());
                assertEquals("Other entity than requested was changed",
                                                TEST_ENTITY1_NAME,
                                                testDataRepository.read(0).get-
Name());
        }

        @Test
        public void testUpdateEntityThatDoesNotExist1() {
                Entity testEntityForUpdatingData = new Entity();

                testEntityForUpdatingData.setId(0);
                testEntityForUpdatingData.setName(TEST_ENTITY3_NAME);

                assertEquals("There was wrong amount of entities in the repos-
itory",
                                                0,
                                                testDataRepository.readAll().enti-
ties.size());

                assertFalse("Update should not have succeeded on non-existing
entity",
                                                testDataRepository.up-
date(testEntityForUpdatingData));


        }

        @Test
        public void testUpdateEntityThatDoesNotExist2() {

                testDataRepository.create(testEntity1);
                Entity testEntityForUpdatingData = new Entity();

                testEntityForUpdatingData.setId(1);
                testEntityForUpdatingData.setName(TEST_ENTITY3_NAME);

                assertEquals("There was wrong amount of entities in the repos-
itory",
                                                1,
```

```
                                                testDataRepository.readAll().enti-
ties.size());

                        assertFalse("Update should not have succeeded on non-existing
entity",
                                                testDataRepository.up-
date(testEntityForUpdatingData));


            }

            @Test
            public void testUpdateEntityThatDoesNotExist3() {

                        testDataRepository.create(testEntity1);
                        testDataRepository.delete(0);
                        testDataRepository.create(testEntity2);
                        Entity testEntityForUpdatingData = new Entity();

                        testEntityForUpdatingData.setId(0);
                        testEntityForUpdatingData.setName(TEST_ENTITY3_NAME);

                        assertEquals("There was wrong amount of entities in the repos-
itory",
                                                1,
                                                testDataRepository.readAll().enti-
ties.size());

                        assertFalse("Update should not have succeeded on non-existing
entity",
                                                testDataRepository.up-
date(testEntityForUpdatingData));

            }

            @Test
            public void testDeleteObjectThatDoesNotExist() {

                        assertFalse("There may not exist an object with the requested
id",
                                                testDataRepository.delete(0));
            }

            @Test
            public void testDeleteEntity() {

                        testDataRepository.create(testEntity1);

                        assertTrue("Entity could not be removed from repository with
requested id",
                                                testDataRepository.delete(0));
            }

            @Test
            public void testDeleteEntititiesFirstInFirstOutOrder() {

                        testDataRepository.create(testEntity1);
                        testDataRepository.create(testEntity2);

                        assertTrue("Entity could not be removed from repository with
requested id",
                                                testDataRepository.delete(0));

                        assertTrue("Entity could not be removed from repository with
requested id",
                                                testDataRepository.delete(1));
            }

            @Test
            public void testDeleteEntitiesLastInFirstOutOrder() {

                        testDataRepository.create(testEntity1);
                        testDataRepository.create(testEntity2);
```

```
                        assertTrue("Entity could not be removed from repository with
requested id",
                                                testDataRepository.delete(1));

                        assertTrue("Entity could not be removed from repository with
requested id",
                                                testDataRepository.delete(0));
            }

            @Test
            public void testCheckIdNumberingAfterCreatingSequentialEntities() {

                        for (int i=0;i<5;i++)
                        {
                                    testDataRepository.create(testEntity1);
                                    assertEquals("Id's were not created in sequential
order",
                                                                        i,
                                                                        testDataRepo-
sitory.read(i).getId());
                        }
            }

            @Test
            public void testCheckIdNumberingAfterCreatingSequnceAndDeletingSequence() {

                        for (int i=0;i<5;i++)
                        {
                                    testDataRepository.create(testEntity1);
                                    testDataRepository.delete(i);
                        }

                        testDataRepository.create(testEntity1);

                        assertEquals("Wrong amount of entities in repository",
                                                1,
                                                testDataRepository.readAll().enti-
ties.size());
                        assertEquals("Next assigned id was not correct",
                                                5,
                                                testDataRepository.readAll().enti-
ties.get(0).getId());
            }

            /*
             * This test fails because the problem
             * is not addressed in DataRepository
             * implementation.
             * This demonstrates the problems
             * mock objects cannot address,
             * but can be caught in integration
             * test.
             * Hence test ignored.
             */
            @Ignore
            @Test
            public void testCreateNullObject()
            {
                        testDataRepository.create(null);
                        // Assertions missing
            }

}
```

src/test/java/engineering/RestServiceArquillianTest.java

```
package engineering;

import javax.inject.Inject;
import javax.ws.rs.core.Response;
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.WebArchive;
import org.junit.Test;
import org.junit.runner.RunWith;

import data.DataRepository;
import model.Entity;
import model.Entities;
import rest.JaxRsActivator;
import rest.RestService;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;

import static org.junit.Assert.*;

/**
 * RestSserviceArquillianTest.java
 * <p>
 * Demonstrates testing of RestService with Arquillian, deploying a archive for test
including all needed dependencies.
 * @author Erkki Keränen
 *
 */
@RunWith(Arquillian.class)
public class RestServiceArquillianTest {

             final static String TEST_ENTITY1_NAME = "Test Name1";
             final static int TEST_ENTITY1_ID = 0;

             final static String TEST_ENTITY2_NAME = "Test Name2";
             final static int TEST_ENTITY2_ID = 1;

             Entity testEntity1;
             Entity testEntity2;

             /*
              * For running these tests, we must empty the
              * Data Repository between tests. Otherwise
              * Dependency injection would give the same
              * @ApplicationScoped repository for all test
              * runs.
              */
             DataRepository testDataRepository;

             @Inject
             RestService testRestService;

             @Deployment
    public static WebArchive createDeployment() {
        return ShrinkWrap.create(WebArchive.class)
            .addClasses(JaxRsActivator.class,
                        RestService.class,
                        DataRepository.class,
                        Entity.class,
                        Entities.class);
    }

             @BeforeClass
```

```
	public static void setUpBeforeClass() throws Exception {
	}

	@AfterClass
	public static void tearDownAfterClass() throws Exception {
	}

	/*
	 * For Arquillian tests, we create the test
	 * data in every run with the create method
	 * of Rest Service.
	 * We can test the effects on the whole
	 * integration from request to HTTP-status.
	 *
	 * A new, empty datarepository is injected
	 * for every test.
	 */

	@Before
	public void setUp() throws Exception {

			testDataRepository = new DataRepository();
			testRestService.setDataRepository(testDataRepository);

			testEntity1 = new Entity();
			testEntity1.setName(TEST_ENTITY1_NAME);
			testEntity1.setId(TEST_ENTITY1_ID);

			testEntity2 = new Entity();
			testEntity2.setName(TEST_ENTITY2_NAME);
			testEntity2.setId(TEST_ENTITY2_ID);
	}

	@After
	public void tearDown() throws Exception {

			testDataRepository = null;
			testEntity1 = null;
			testEntity2 = null;
	}

	@Test
	public void testRestServiceReadAllStatus200Ok() {

			Response testResponse;

			testResponse = testRestService.readAll();

			assertEquals("Status was not 200 Ok",
					200,
					testResponse.getStatus());
	}


	/*
	 * This test verifies that the repository
	 * is empty in beginning of test run.
	 */
	@Test
	public void testRestServiceReadAllFromEmptyRepositoryReturnsZeroAnd200Ok()
	{
			Response testResponse;

			testResponse = testRestService.readAll();

			Entities testResponseEntities = (Entities) testResponse.getEn-
tity();

			assertEquals("Wrong amount of entities in response",
					0,
					testResponseEntities.enti-
ties.size());
			assertEquals("Status was not 200 OK",
					200,
```

```
                                                testResponse.getStatus());
                }

                @Test
                public void testRestServiceCreateStatus201Created()
                {
                        Response testResponse;

                        testResponse = testRestService.create(testEntity1);
                        assertEquals("Status was not 201 Created",
                                                201,
                                                testResponse.getStatus());
                }

                @Test
                public     void    testRestServiceCreateOneThenReadAllReturnEntitiesWithOn-
eEntity()
                {
                        Response testResponse;

                        testResponse = testRestService.create(testEntity1);
                        assertEquals("Status was not 201 Created",
                                                201,
                                                testResponse.getStatus());

                        testResponse = null;
                        testResponse = testRestService.readAll();

                        Entities testResponseEntities = (Entities) testResponse.getEn-
tity();

                        assertEquals("Wrong amount of entities in response",
                                                1,
                                                testResponseEntities.enti-
ties.size());
                        assertEquals("Status was not 200 OK",
                                                200,
                                                testResponse.getStatus());
                }

                @Test
                public void testRestServiceCreateOneReaditWithReadAllMethod()
                {
                        Response testResponse;

                        testResponse = testRestService.create(testEntity1);
                        assertEquals("Status was not 201 Created",
                                                201,
                                                testResponse.getStatus());

                        testResponse = null;
                        testResponse = testRestService.readAll();

                        Entities testResponseEntities = (Entities) testResponse.getEn-
tity();

                        assertEquals("Wrong amount of entities in response",
                                                1,
                                                testResponseEntities.enti-
ties.size());
                        assertEquals("Wrong id",
                                                TEST_ENTITY1_ID,
                                                testResponseEntities.enti-
ties.get(0).getId());
                        assertEquals("Wrong name",
                                                TEST_ENTITY1_NAME,
                                                testResponseEntities.enti-
ties.get(0).getName());
                        assertEquals("Status was not 200 OK",
                                                200,
                                                testResponse.getStatus());
                }

                @Test
```

```
		public void testRestServiceCreateOneThenReadIt()
		{
				Response testResponse;

				testResponse = testRestService.create(testEntity1);
				assertEquals("Status was not 201 Created",
										201,
										testResponse.getStatus());

				testResponse = null;
				testResponse = testRestService.read(0);

				Entity testResponseEntity = (Entity) testResponse.getEntity();

				assertEquals("Wrong id",
										TEST_ENTITY1_ID,
										testResponseEntity.getId());
				assertEquals("Wrong name",
										TEST_ENTITY1_NAME,
										testResponseEntity.getName());
				assertEquals("Status was not 200 OK",
										200,
										testResponse.getStatus());
		}

		@Test
		public void testRestServiceReadNonExisting404NotFound()
		{
				Response testResponse;

				testResponse = testRestService.read(1);
				assertEquals("Status was not 404 Not Found",
										404,
										testResponse.getStatus());
		}

		/*
		 * Compared to Mock-based testing,
		 * in this integration test following
		 * will fail in NullPointerException.
		 *
		 * So maybe we missed testing this
		 * issue in the Datarepository tests?
		 *
		 * (The assertion is also wrong)
		 *
		 */
		@Ignore
		@Test
		public void testRestServiceCreateEntityNullObject()
		{
				Response testResponse;

				testResponse = testRestService.create(null);
				assertEquals("Status was not 201 Created",
										201,
										testResponse.getStatus());
		}

		@Test
		public void testRestServiceCreateEntityThenDeleteEntityStatus204NoContent()
		{
				Response testResponse;

				testResponse = testRestService.create(testEntity1);
				assertEquals("Status was not 201 Created",
										201,
										testResponse.getStatus());

				testResponse = null;
				testResponse = testRestService.readAll();
				Entities testResponseEntities = (Entities) testResponse.getEn-
tity();
```

```
                        assertEquals("Wrong number of entities in response",
                                                1,
                                                testResponseEntities.enti-
ties.size());

                        testResponse = null;
                        testResponse = testRestService.delete(0);
                        assertEquals("Status was not 204 No Content",
                                                204,
                                                testResponse.getStatus());

            }

            @Test
            public  void  testRestServiceCreateEntityThenDeleteNonExistingStatus404Not-
Found1()
            {
                        Response testResponse;

                        testResponse = testRestService.create(testEntity1);
                        assertEquals("Status was not 201 Created",
                                                201,
                                                testResponse.getStatus());

                        testResponse = null;
                        testResponse = testRestService.readAll();
                        Entities testResponseEntities = (Entities) testResponse.getEn-
tity();
                        assertEquals("Wrong number of entities in response",
                                                1,
                                                testResponseEntities.enti-
ties.size());

                        testResponse = null;
                        testResponse = testRestService.delete(1);
                        assertEquals("Status was not 404 Not Found",
                                                404,
                                                testResponse.getStatus());

            }

            @Test
            public  void  testRestServiceCreateEntityThenDeleteNonExistingStatus404Not-
Found2()
            {
                        Response testResponse;

                        testResponse = testRestService.create(testEntity1);
                        assertEquals("Status was not 201 Created",
                                                201,
                                                testResponse.getStatus());

                        testResponse = null;
                        testResponse = testRestService.readAll();
                        Entities testResponseEntities = (Entities) testResponse.getEn-
tity();
                        assertEquals("Wrong number of entities in response",
                                                1,
                                                testResponseEntities.enti-
ties.size());

                        testResponse = null;
                        testResponse = testRestService.delete(-1);
                        assertEquals("Status was not 404 Not Found",
                                                404,
                                                testResponse.getStatus());

            }

            @Test
            public void testRestServiceCreateOneThenUpdateItStatus200()
            {
                        Response testResponse;
```

```
                        testResponse = testRestService.create(testEntity1);
                        assertEquals("Status was not 201 Created",
                                                201,
                                                testResponse.getStatus());

                        testResponse = null;
                        testResponse = testRestService.readAll();
                        Entities testResponseEntities = (Entities) testResponse.getEn-
tity();
                        assertEquals("Wrong number of entities in response",
                                                1,
                                                testResponseEntities.enti-
ties.size());

                        Entity testEntityForUpdateOperation = new Entity();

                        testEntityForUpdateOperation.setId(0);
                        testEntityForUpdateOperation.setName("Updated Name");

                        testResponse = null;
                        testResponse      =      testRestService.update(testEntityFo-
rUpdateOperation);
                        assertEquals("Status was not 200 Ok",
                                                200,
                                                testResponse.getStatus());

                        testResponse = null;
                        testResponse = testRestService.read(0);
                        Entity testResponseEntity = (Entity) testResponse.getEntity();
                        assertEquals("Wrong id",
                                                0,
                                                testResponseEntity.getId());
                        assertEquals("Wrong name",
                                                testEntityForUpdateOperation.get-
Name(),
                                                testResponseEntity.getName());


            }

            @Test
            public  void  testRestServiceCreateOneUpdateThenNonExistingStatus400BadRe-
quest()
            {
                        Response testResponse;

                        testResponse = testRestService.create(testEntity1);
                        assertEquals("Status was not 201 Created",
                                                201,
                                                testResponse.getStatus());

                        testResponse = null;
                        testResponse = testRestService.readAll();
                        Entities testResponseEntities = (Entities) testResponse.getEn-
tity();
                        assertEquals("Wrong number of entities in response",
                                                1,
                                                testResponseEntities.enti-
ties.size());

                        Entity testEntityForUpdateOperation = new Entity();

                        testEntityForUpdateOperation.setId(3);
                        testEntityForUpdateOperation.setName("Updated Name");

                        testResponse = null;
                        testResponse      =      testRestService.update(testEntityFo-
rUpdateOperation);
                        assertEquals("Status was not 400 Bad Request",
                                                400,
                                                testResponse.getStatus());
            }
}
```

src/test/java/engineering/RestServiceJUnitTest.java

```
package engineering;

import static org.junit.Assert.*;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.verify;

import javax.ws.rs.core.Response;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import data.DataRepository;
import model.Entities;
import model.Entity;
import rest.RestService;

/**
 * RestServiceJUnitTest.java
 * <p>
 * Demonstrates unit testing of RestService using mock objects to mock the behavior of
dependencies.
 * @author Erkki Keränen
 *
 */
public class RestServiceJUnitTest {

		RestService testRestService;
		DataRepository mockDataRepository;

		Entities testEntitiesWithOneEntity;

		Entity testEntity1,
		                           testEntity2;

		final static String TEST_ENTITY1_NAME = "Test Nimi1";
		final static int TEST_ENTITY1_ID = 0;

		final static String TEST_ENTITY2_NAME = "Test Nimi2";
		final static int TEST_ENTITY2_ID = 1;

		@BeforeClass
		public static void setUpBeforeClass() throws Exception {
		}

		@AfterClass
		public static void tearDownAfterClass() throws Exception {
		}

		/*
		 * With mocking, the assumed data
		 * must be created, so the functionality
		 * can be inspected
		 */

		@Before
		public void setUp() throws Exception {

				testEntitiesWithOneEntity = new Entities();
				testEntity1 = new Entity();
				testEntity1.setName(TEST_ENTITY1_NAME);
				testEntity1.setId(TEST_ENTITY1_ID);
				testEntitiesWithOneEntity.entities.add(testEntity1);

				testEntity2 = new Entity();
```

```
                testEntity2.setName(TEST_ENTITY2_NAME);
                testEntity2.setId(TEST_ENTITY2_ID);

                mockDataRepository = mock(DataRepository.class);
                when(mockDataRepository.readAll()).
                                thenReturn(testEntitiesWithOneEntity);

                when(mockDataRepository.read(0)).
                                thenReturn(testEntity1);

                when(mockDataRepository.read(1)).
                                thenReturn(null);

                when(mockDataRepository.delete(0)).
                                thenReturn(true);

                when(mockDataRepository.delete(1)).
                                thenReturn(false);

                when(mockDataRepository.update(testEntity1)).
                thenReturn(true);

                when(mockDataRepository.update(testEntity2)).
                                thenReturn(false);

                testRestService = new RestService();

                testRestService.setDataRepository(mockDataRepository);

        }

        @After
        public void tearDown() throws Exception {
                testRestService = null;
                mockDataRepository = null;

                testEntitiesWithOneEntity = null;
                testEntity1 = null;
                testEntity2 = null;
        }

        @Test
        public void testRestServiceCreateEntityStatus201Created() throws Exception{

                Response testResponse = testRestService.create(testEntity2);
                assertEquals("Status was not 201 Created",
                                                201,
                                                testResponse.getStatus());
                verify(mockDataRepository).create(testEntity2);
        }

        /*
         * The following test cannot
         * be tested reliably with
         * mocks. Compare to Arquillian
         * integration test that throws
         * NullPointerException
         *
         * The outcome of this kind of
         * tests with mocking is
         * questionable.
         *
         * The assertion is also
         * based on false assumptions
         * and should be corrected.
         *
         */

        @Test
        public void testRestServiceCreateNullObject() throws Exception{

                Response testResponse = testRestService.create(null);
                assertEquals("Status was not 201 Created",
                                                201,
```

```
                                        testResponse.getStatus());
                        verify(mockDataRepository).create(null);
            }

            @Test
            public  void  testRestServiceReadAllReturnsTestEntityInEntitiesStatus200Ok()
throws Exception{

                        Response testResponse = testRestService.readAll();
                        Entities testResponseEntities = (Entities) testResponse.getEn-
tity();
                        assertEquals("Wrong number of entities was returned",
                                        1,
                                        testResponseEntities.enti-
ties.size());
                        assertEquals("Entity had wrong id",
                                        TEST_ENTITY1_ID,
                                        testResponseEntities.enti-
ties.get(0).getId());
                        assertEquals("Entity had wrong name",
                                        TEST_ENTITY1_NAME,
                                        testResponseEntities.enti-
ties.get(0).getName());
                        assertEquals("Status was not 200 OK",
                                        200,
                                        testResponse.getStatus());
                        verify(mockDataRepository).readAll();
            }

            @Test
            public void testRestServiceReadReturnsTestEntityStatus200Ok() throws Excep-
tion{
                        Response testResponse = testRestService.read(0);

                        Entity testResponseEntity = (Entity) testResponse.getEntity();
                        assertEquals("Entity had wrong id",
                                        TEST_ENTITY1_ID,
                                        testResponseEntity.getId());
                        assertEquals("Entity had wrong name",
                                        TEST_ENTITY1_NAME,
                                        testResponseEntity.getName());
                        assertEquals("Status was not 200 Ok",
                                        200,
                                        testResponse.getStatus());
                        verify(mockDataRepository).read(0);
            }

            @Test
            public void testRestServiceReadNonExistingStatus404NotFound() throws Excep-
tion{
                        Response testResponse = testRestService.read(1);

                        Entity testResponseEntity = (Entity) testResponse.getEntity();
                        assertEquals("Body should not have content",
                                        null,
                                        testResponseEntity);
                        assertEquals("Status was not 404 Not Found",
                                        404,
                                        testResponse.getStatus());
                        verify(mockDataRepository).read(1);
            }

            @Test
            public void testRestServiceUpdateExistingStatus200Ok() throws Exception{
                        Response testResponse = testRestService.update(testEntity1);
                        assertEquals("Status was not 200 Ok",
                                        200,
                                        testResponse.getStatus());
                        verify(mockDataRepository).update(testEntity1);
            }

            @Test
            public  void  testRestServiceUpdateNonExistingStatus400BadRequest()  throws
Exception{
```

```
                        Response testResponse = testRestService.update(testEntity2);
                        assertEquals("Status was not 400 Bad Request",
                                                400,
                                                testResponse.getStatus());
                        verify(mockDataRepository).update(testEntity2);
                }

                @Test
                public void testRestServiceDeleteExistingStatus204NoContent() throws Excep-
tion{
                        Response testResponse = testRestService.delete(0);
                        assertEquals("Status was not 204 No Content",
                                                204,
                                                testResponse.getStatus());
                        verify(mockDataRepository).delete(0);
                }

                @Test
                public void testRestServiceDeleteNonExistingStatus404NotFound() throws Ex-
ception{
                        Response testResponse = testRestService.delete(1);
                        assertEquals("Status was not 404 Not Found",
                                                404,
                                                testResponse.getStatus());
                        verify(mockDataRepository).delete(1);
                }


}
```