Accesing Databases with JPA

Introduction to Service Design and Engineering 2013/2014.
Lab session #8
University of Trento

Outline
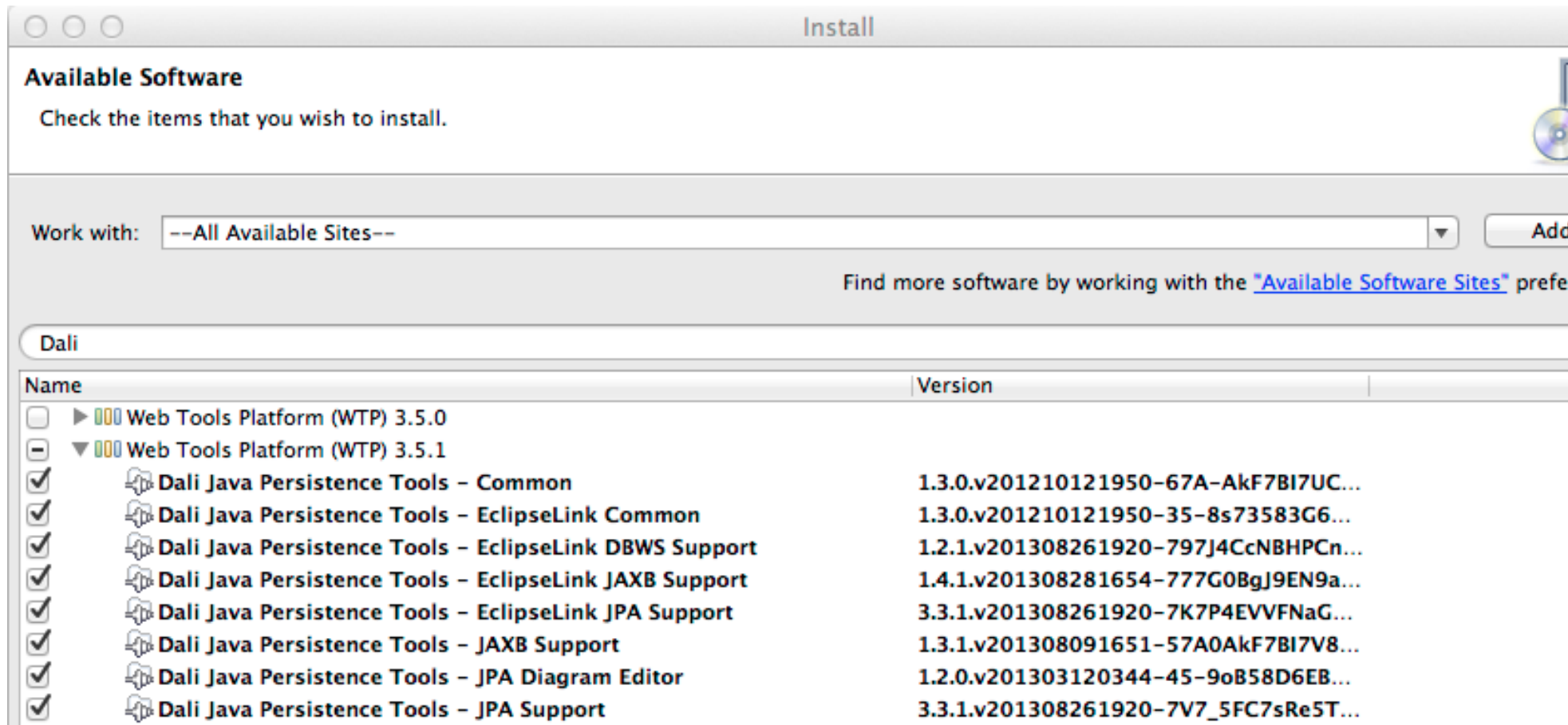
- JPA Overview
- JPA Example
- CRUD Restful with JPA support

Pre-Requisites (1):

- Download and Install SQLite
- Download the SQLite JDBC driver
- Download and Install H2 (all platforms version, the jar inside bin is the jdbc driver)
- JAXB and Jersey libraries from previous lab sessions
- Download SQLite Studio (we will use it to create/explore an sqlite DB)

Pre-Requisites (2):

- JPA Support:
  - Eclipse: make you sure to have the **Dali Java Persistence Tools**
  - Without Eclipse: [Download EclipseLink jars](#) -> eclipselink.jar, javax.persistence_*

JPA Overview (1)

- JPA stands for **Java Persistence API**
- JPA is a **Java specification** for **ORM** (another is JDO).
  - ORM stands for **Object-relational mapping:** the process of mapping objects to relational tables (and vice versa).
  - ORM allow developers to **work directly with objects** rather then with SQL statements.
  - The JPA implementation is typically called **persistence provider**
  - Some implementations are: Hibernate, EclipseLink and Apache OpenJPA

JPA Overview (2)

- Mapping between Java objects and database tables is defined via **persistence metadata**.
- JPA providers use persistence metadata to perform the correct database operations.
- Persistence metadata is tipically specified via **annotations in the Java class**.
  - Alternatively the metadata can be defined via XML files or a combination of both. A XML configuration overwrites the annotations (see example)

Tutorial JPA: Simple project (1)

- Create a new JPA project in Eclipse
  - If you do not use eclipse, all you need is EclipseLink libraries you downloaded bef
  - In your source folder (e.g., src) create a folder named **"META-INF"**
  - Add a file named **"persistence.xml"** in this folder.
- Add the JDBC SQLite Driver to the classpath of the project
- Open the **persistence.xml**. You should see the following, with the persistence-unit nam

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence data-brackets-id='248' version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    <persistence-unit data-brackets-id='249' name="introsde-jpa">
    </persistence-unit>
</persistence>
```

Tutorial JPA: Simple project (1)

- Create a new JPA project in Eclipse
  - If you do not use eclipse, all you need is EclipseLink libraries you downloaded before to be in the classpath of your project
  - In your source folder (e.g., src) create a folder named **"META-INF"**
  - Add a file named **"persistence.xml"** in this folder.
- Add the JDBC SQLite Driver to the classpath of the project
- Open the **persistence.xml**. You should see the following, with the persistence-unit name equal to the name of your project (if not, add it)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence data-brackets-id='250' version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit data-brackets-id='251' name="introsde-jpa">
    </persistence-unit>
</persistence>
```

Tutorial JPA: Adding Models (1)

- Open the **lifecoach.sqlite** in Sqlite Studio and explore the schema.
- After you know very well what tables are in the database and how they relate to each other, the next step is to create java classes that will represent this data model. **This is what we call the MODEL**
- Create a package named **model** and add the first model for the table **Person**

```java
import java.io.Serializable;
import javax.persistence.*;
@Entity  // indicates that this class is an entity to persist in DB
@Table(name="Person") // to whate table must be persisted
public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id // defines this attributed as the one that identifies the entity
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="idPerson") // maps the following attribute to a column
    private int idPerson;
    @Column(name="lastname")
    private String lastname;
    @Column(name="name")
    private String name;
    @Column(name="username")
    private String username;
    @Temporal(TemporalType.DATE) // defines the precision of the date attribute
    @Column(name="birthdate")
    private Date birthdate;
    @Column(name="email")
    private String email;
    // getters and setters of all the private attributes
```

Tutorial JPA: Adding Models (2)

- Classes that will be persisted in a database must be annotated with **@Entity** annotation.
- By default, the table name corresponds to the class name. You can change this with the addition to the annotation **@Table(name="NEWTABLENAME")**.
- An entity represents a *table* in the database. Instances of the class will represents its *rows*.
- Fields of the Entity will be saved in the database and JPA can use either instance variables (fields) or the corresponding getters and setters to access the fields. **It is not allowed to mix both methods**.
- By default each field is mapped to a column with the name of the field. You can change the default name via **@Column (name="newColumnName").**

Tutorial JPA: Adding Models (3)

- The following annotations can be used in fields:
    - **@Id:** Identifies the unique ID of the database entry
    - **@GeneratedValue:** Together with ID defines that this value is generated automatically.
    - **@Transient** Field will not be saved in database

Exercise 1: Creating JPA Models

- Create the model for the *LifeStatus* table in Lifecoach DB

Tutorial JPA: Accessing the Database (1)

- Create a package named **dao (data access objects)** and the following class to it.

```java
package dao;
import java.util.List;
import javax.persistence.*;
import model.Person;
public enum LifeCoachDao {
    instance;
    private EntityManagerFactory emf;
    private LifeCoachDao() {
        if (emf!=null) {
            emf.close();
        }
        emf = Persistence.createEntityManagerFactory("introsde-jpa");
    }
    public EntityManager createEntityManager() {
        return emf.createEntityManager();
    }
    public void closeConnections(EntityManager em) {
        em.close();
    }
    public EntityTransaction getTransaction(EntityManager em) {
        return em.getTransaction();
    }
    public EntityManagerFactory getEntityManagerFactory() {
        return emf;
    }
}
```

Tutorial JPA: Operations in the Database (1)

- The **LifeCoachDao** is a singleton java instance that contains an **EntityManagerFactory**, which is configured by the persistence unit **"introsde-jpa"**
- This class will be used to create and **Entity Manager** whenever we need to execute an operation in the Database.
- The entity manager provides the operations from and to the database,
  - e.g. find objects, persists them, remove objects from the database, etc.
- In JavaEE applications, the entity manager is automatically inserted in the web application. Outside JavaEE you need to manage the entity manager yourself.

Tutorial JPA: Operations in the Database (2)

- To save objects in the database, the Entity Manager provides the method **save()**
- To synchronize objects again with the database a Entity Manager provides the **merge()** method.
- If the Entity Manager is closed (via **close()**) then the managed entities are in a detached state.

Tutorial JPA: Operations in the Database (3)

- Add the following methods to the Person model

```java
public class Person {
    // attributes and other methods...
    public static Person getPersonById(int personId) {
        EntityManager em = LifeCoachDao.instance.createEntityManager();
        Person p = em.find(Person.class, personId);
        LifeCoachDao.instance.closeConnections(em);
        return p;
    }
    public static List<Person> getAll() {
        EntityManager em = LifeCoachDao.instance.createEntityManager();
        List<Person> list = em.createNamedQuery("Person.findAll", Person.class)
                .getResultList();
        LifeCoachDao.instance.closeConnections(em);
        return list;
    }
    public static Person savePerson(Person p) {
        EntityManager em = LifeCoachDao.instance.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        em.persist(p);
        tx.commit();
        LifeCoachDao.instance.closeConnections(em);
        return p;
    }
    // continues in the next page
}
```

Tutorial JPA: Operations in the Database (4)

```java
public class Person {
    // previous page code
    public static Person updatePerson(Person p) {
        EntityManager em = LifeCoachDao.instance.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        p=em.merge(p);
        tx.commit();
         LifeCoachDao.instance.closeConnections(em);
         return p;
         }
    public static void removePerson(Person p) {
         EntityManager em = LifeCoachDao.instance.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();
         p=em.merge(p);
         em.remove(p);
         tx.commit();
        LifeCoachDao.instance.closeConnections(em);
    }
}
```

Tutorial JPA: Testing the connection (1)

- Add JUnit support to your project
  - In Eclipse: *Project -> Build Path -> Configure Build Path -> Add Library -> JUnit*
  - Without IDE: download the jars from the [JUnit Website](#)
- Create a package named **test**

Tutorial JPA: Testing the connection (2)

- Create the following junit test class in the test package and run it as a JUnit class

```java
package test;
import static org.junit.Assert.*;
import model.Person;
import org.junit.Test;
import dao.LifeCoachDao;
import java.util.List;
public class JPAStarterTest {
  @Test
  public void readPerson() {
      Person p = Person.getPersonById(1);
      assertEquals("Table has correct name", "Chuck", p.getName());
      List<Person data-brackets-id='254'> list = Person.getAll();
      assertEquals("Table has one entity", 1, list.size());
  }
  // continues in the next page
```

Tutorial JPA: Testing the connection (3)

```java
@Test
public void addPersonWithDao() {
    Person p = new Person();
    p.setName("Pinco");
    p.setLastname("Pallino");
    Person.savePerson(p);
    assertNotNull("Id should not be null", p.getIdPerson());
    List<Person data-brackets-id='255'> list = Person.getAll();
    assertEquals("Table has two entities", 2, list.size());
    assertEquals("Table has correct name", "Pinco", list.get(1).getName());
    Person created = list.get(1);
    Person.removePerson(created);
    list = Person.getAll();
    assertEquals("Table has two entities", 1, list.size());
    assertEquals("Table has correct name", "Chuck", list.get(0).getName());
}
```

Tutorial JPA: SQLite sequences sidenote

- SQLITE implements auto increment ids through named sequences that are stored in a special table named "sqlite_sequence"
- For this reason, you need to use the following @GeneratedValue annotation

```java
@GeneratedValue(generator="sqlite_person")
@TableGenerator(name="sqlite_person", table="sqlite_sequence",
    pkColumnName="name", valueColumnName="seq",
    pkColumnValue="Person")
@Column(name="idPerson")
```

Tutorial JPA: Generating Entities (1)

- This part of the tutorial is **for eclipse only**
  - If not using eclipse, jump to the part of mapping relationships with JPA
- Open data source view and add a new conection (Right Click on *Database connections -> New..*)
- **Add the correct SQLite Driver**
  - Click on the add button right of the "Drivers" select
  - Select "SQLite JDBC Driver"
  - If eclipse is unable to locat the jar of the driver, go to the tab *JAR List*, remove the current jar and add the SQLite JDBC driver you have downloaded
- **Configure the connection**
  - Database = lifecoach
  - Database Location = PATH_TO_YOUR_LOCAL_LIFECOACH_SQLITE_FILE
- Right click on your project and select **JPA tools --> Generate Entities from Tables**

Tutorial JPA: Generating Entities (2)

- Select the connection, get the list of tables and select the tables that you want to use to generate entities

Tutorial JPA: Generating Entities (3)

- The second step for generating entities is to specify **Table Associations** between entities to map relationship between tables. Add a relationship between *LifeStatus* and *MeasureDefinition*

Tutorial JPA: Generating Entities (4)

- Specify the columns that define the association

Tutorial JPA: Generating Entities (5)

- Specify the cardinality of the association

Tutorial JPA: Generating Entities (6)

- Finally, you can choose whether which entities in the association should have a property referencing the other entity (it is not required that both have a property referencing the other entity)

Tutorial JPA: Generating Entities (7)

- The next step to select some defaults properties for the mapping, like the default key generator. Since we are using SQLite and there is a special generator for this, let's choose "None"

Tutorial JPA: Generating Entities (8)

- The last step is to define how each property will be mapped (name of the attributes int the model classes, type of each attribute, etc.). Make sure all the primary keys have the mapping kind to "id".

Tutorial JPA: Relationships (1)

- You should have now all the entity classes generated in the model package.
  - If you didn't use the generation wizard, copy the model classes from the introsde-jp
    in this lab code
  - In generated classes, you can delete \"
- Open the LifeStatus entity. You should have the following attribute (if not, added). This
  map a @ManyToOne relationship between LifeStatus and MeasureDefinition:

```
@ManyToOne
@JoinColumn(name = "idMeasureDef", referencedColumnName = "idMeasureDef", insertable = true, updat
private MeasureDefinition measureDefinition;
```

Tutorial JPA: Adding Relationships (2)

- JPA allows to define relationships between classes, e.g. it can be defined that a class is part of another class (containment).
- Classes can have one to one, one to many, many to one, and many to many relationships with other classes.
- A relationship can be bidirectional or unidirectional, e.g. in a bidirectional relationship both classes store a reference to each other while in an unidirectional case only one class has a reference to the other class.
- Within a bidirectional relationship you need to specify the owning side of this relationship in the other class with the attribute "mappedBy", e.g. **@ManyToMany(mappedBy="attributeOfTheOwningClass".**
- JPA Relationship annotations:

```
@OneToOne
@OneToMany
@ManyToOne
@ManyToMany
```

Exercise 3:

- Control the code of the models and:
    - add the relationships that are missing in all the models
    - for each class, add the correct @GeneratedValue annotations following the example of Person

Exercise 4:

- Now that you have all the models of the database, add CRUD operations to all of them like the ones in Person (i.e., using transactions to persist, merge and remove entities in the database)

Tutorial JPA: Testing Relationships

- Add the following testing method to your testing class and check that when you update a person with a new LifeStatus measure, this is effectively added in the database.

```java
@Test
public void testLifeStatusPersonRelationship() {
    // setting weight for an existing person with existing measures
    Person chuck = Person.getPersonById(1);
    assertEquals("Chuck norris is here", "Chuck", chuck.getName());
    // add a new measure value to the list of measurements of chuck
    MeasureDefinition md = MeasureDefinition
        .getMeasureDefinitionById(1);
    LifeStatus l = new LifeStatus();
    l.setMeasureDefinition(md);
    l.setValue("85");
    chuck.getLifeStatus().add(l);
    chuck = Person.updatePerson(chuck);
    assertEquals("Person should have now two measures", 2, chuck.getLifeStatus().size());
    l=chuck.getLifeStatus().get(1);
    assertNotNull("LifeStatus measure was created", l.getIdMeasure());
    chuck.getLifeStatus().remove(1);
    Person.updatePerson(chuck);
    assertEquals("Person should have now just one measure", 1, chuck.getLifeStatus().size());
}
```

Tutorial Jersey+JPA: Adding Resources (1)

- Now that we can connect to a Database, the next step is to use this model and connection in REST resources that are exposed using Jersey.
- The first step is to add Jersey support to your project
  - In Eclipse: Project Properties -> Project Facets -> Enable Dynamic Web Module
  - Without Eclipse: create a WebContent folder in the root of the project with a subdirectory named WEB-INF and a web.xml as the one below inside.
  - Add Jersey support by adding the Jersey-Bundle libraries to the build path

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app data-brackets-id='256' id="WebApp_ID" version="3.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
</web-app>
```

Tutorial Jersey+JPA: Adding Resources (2)

- Create a package **resources** for the resources of your REST API.
- Copy from lab7 the following resources:
  - PeopleResource
  - PersonResource
  - StandaloneServer (remember to change project properties as indicated in the comments of this class)
- Fix the errors
  - i.e., import the correct model classes, use and int id instead of String
- Replace the PersonDao references with calls to the static methods in the models
  - i.e., *Person.getPersonById()* to read a person, *Person.savePerson(p)* to save a person, *Person.getAll()* to get the whole list of people in the database

Tutorial Jersey+JPA: Persistance methods in models (3)

- Examples of how to replace PersonDao References to calls to the methods in models (for PeopleResource)

```java
@Path("/person")
public class PeopleResource {
    // attributes and other methos in People resource
    @GET
    @Produces(MediaType.TEXT_XML)
    public List<Person data-brackets-id='257'> getPersonsBrowser() {
        List<Person data-brackets-id='258'> people = Person.getAll();
        return people;
    }
    @POST
    @Produces(MediaType.APPLICATION_XML)
    @Consumes(MediaType.APPLICATION_XML)
    public Person newPerson(Person person) throws IOException {
        System.out.println("Creating new person...");
        person = Person.savePerson(person);
        return person;
    }
}
```

Tutorial Jersey+JPA: Persistance methods in models (4)

- Examples of how to replace PersonDao References to calls to the methods in models (for PersonResource)

```java
public class PersonResource {
    // attributes and other methos in Person resource
    @GET
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
    public Person getPerson() {
        Person person = this.getPersonById(id);
        if (person == null)
            throw new RuntimeException("Get: Person with " + id + " not found");
        return person;
    }
    @DELETE
    public void deletePerson() {
        Person c = Person.getPersonById(id);
        if (c == null)
          throw new RuntimeException("Delete: Person with " + id
                  + " not found");
        Person.removePerson(c);
    }
    public Person getPersonById(int personId) {
        System.out.println("Reading person from DB with id: "+personId);
        Person person = Person.getPersonById(personId);
        return person;
    }
}
```

Tutorial Jersey+JPA: Enabling JAXB

- The last step is to add JAXB Annotations to the model classes to allow their proper serialization to and from xml
- **Example:**
    - Add **@XmlRootElement** on top of the Person model
    - Add **@XmlRootElement(name="Measure")** on top of the LifeStatus model
    - The person element on LifeStatus should not be serialized (to avoid infinites loops because lifestatus is serialized within Person), so add **@XmlTransient** on the getter of person in the LifeStatus class
    - To serialize each LifeStatus instance add the following to the LifeStatus getter on Person

```
@XmlElementWrapper(name = "Measurements")
@XmlElement(name = "Measure")
```

Exercise 5:

- Fix the remaining services so that all the services use the methods provided by models read/create/update/delete objects in the database.
- Add JAXB annotations to the models that do not have them yet.

Exercise 6 (1):

- Run the standalone server and test the services using [postman client](#)
    - GET localhost:5900/person (should give you the list)
    - GET localhost:5900/person/1 (should give you the record of the only person in the DB
    - Try POST localhost:5900/person with the following data in the body

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person data-brackets-id='259'>
    <email data-brackets-id='260'>pinco.pallino@gmail.com</email>
    <lastname data-brackets-id='261'>Pinco</lastname>
    <name data-brackets-id='262'>Pallino</name>
    <username data-brackets-id='263'>pinco</username>
</person>
```

Exercise 6 (2):

- Try also the Update
  - PUT localhost:5900/person/1

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person data-brackets-id='264'>
  <birthdate data-brackets-id='265'>1945-01-01T00:00:00+01:00</birthdate>
  <email data-brackets-id='266'>chuck.norris@gmail.com</email>
  <idPerson data-brackets-id='267'>1</idPerson>
  <lastname data-brackets-id='268'>Norris</lastname>
  <Measurements data-brackets-id='269'>
    <Measure data-brackets-id='270'>
      <idMeasure data-brackets-id='271'>1</idMeasure>
      <measureDefinition data-brackets-id='272'>
          <idMeasureDef data-brackets-id='273'>1</idMeasureDef>
          <measureName data-brackets-id='274'>weight</measureName>
          <measureType data-brackets-id='275'>double</measureType>
      </measureDefinition>
      <value data-brackets-id='276'>86</value>
    </Measure>
     </Measurements>
  <name data-brackets-id='277'>Chuck</name>
  <username data-brackets-id='278'>chuck</username>
</person>
```

Lab session source code

- All the source code covered in this session (which is also the solution to the exercises) is part of the introsde-jpa project available in this lab's root folder.
- The project contains some other features that are left to you to discover.

Other Resources

- JUnit Tutorial
- JPA tutorial from where we took some of the explanations
- Checkout also mashape.com and signup with your GitHub account (we will try to use an API from there in the future sessions)