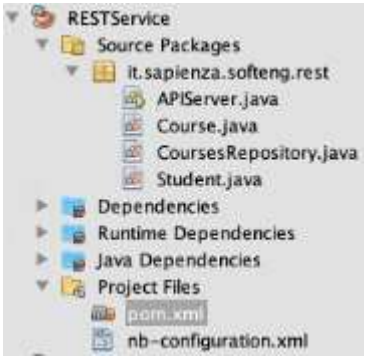
LAB 5 - REST

POSTman simulates a REST client(useful to test rest servers)



WebService

pom should work.

In this example data is exchange (marshalled and unmarshalled) in xml.

small notes: @Path("serviceBaseFolder") specifies the first parameter that identifies the entire service in a domain; @Produces("text/xml") specifies the type of response (there can be more than one); @GET/@PUT/@POST/@DELETE specify the API, if followed by a @Path("...") then it appends it after the method (basically a parameter for the method.

- Student: this class is called a dto class (data transfer obejct) and has no functionality. It's role is to only be a data structure, and has only the role of sending data back and forth. It has the properties and methods (set and get for each property), and then java prescribes the overriding of the equals method (and hashCode() too if necessary). Note the "@XmlRootElement": this tells the library that this class provides the marshaling for the tag "Student".

```java
package it.sapienza.softeng.rest;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "Student")
public class Student {
    private int id;
    private String name;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public int hashCode() {
        return id + name.hashCode();
    }
    @Override
    public boolean equals(Object obj) {
        return (obj instanceof Student) && (id == ((Student) obj).getId()) && (name.equals(((Student) obj).getName()));
    }
}
```

- Course: this class is not only a tag, but also contains other tags. still has the @ that tells this is the reference for marshaling course objects. As before it has the properties and the set/get methods for them. On top of that, it exposes the API using the keywords @GET, @POST etc. that implement the appropriate logic. Notice the private method findById

```
@GET
@Path("{studentId}")
public Student getStudent(@PathParam("studentId") int studentId) {
    return findById(studentId);
}

@POST
public Response createStudent(Student student) {
    for (Student element : students) {
        if (element.getId() == student.getId()) {
            return Response.status(Response.Status.CONFLICT).build();
        }
    }
    students.add(student);
    return Response.ok(student).build();
}

@DELETE
@Path("{studentId}")
public Response deleteStudent(@PathParam("studentId") int studentId) {
    Student student = findById(studentId);
    if (student == null) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }
    students.remove(student);
    return Response.ok().build();
}
```

- CourseRepository: It's the root repo of our service (@Path and @produces keywords). It maintains a data structure(HashMap over int id and obj course) for manging all the courses . It only has get and put active. Then it also has a "subroute" "{courseid}/students" that tells how to move from the courses to the students in the given course. Also contains any data to initialise the server data (Static allocation).

```
package it.sapienza.softeng.rest;
import javax.ws.rs.*;
import javax.ws.rs.core.Response;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Path("/courses")
@Produces("text/xml")
public class CoursesRepository {

    private Map<Integer, Course> courses = new HashMap<>();
    {
        Student student1 = new Student();
        Student student2 = new Student();
```

```
        student1.setId(1);
        student1.setName("Student A");
        student2.setId(2);
        student2.setName("Student B");

        List<Student> course1Students = new ArrayList<>();
        course1Students.add(student1);
        course1Students.add(student2);

        Course course1 = new Course();
        Course course2 = new Course();
        course1.setId(1);
        course1.setName("REST with Spring");
        course1.setStudents(course1Students);
        course2.setId(2);
        course2.setName("Software Engineering");

        courses.put(1, course1);
        courses.put(2, course2);
    }
```

```
@GET
@Path("{courseId}")
public Course getCourse(@PathParam("courseId") int courseId) {
    return findById(courseId);
}

@PUT
@Path("{courseId}")
public Response updateCourse(@PathParam("courseId") int courseId, Course course) {
    Course existingCourse = findById(courseId);
    if (existingCourse == null) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }
    if (existingCourse.equals(course)) {
        return Response.notModified().build();
    }
    courses.put(courseId, course);
    return Response.ok().build();
}

@Path("{courseId}/students")
public Course pathToStudent(@PathParam("courseId") int courseId) {
    return findById(courseId);
}

private Course findById(int id) {
    for (Map.Entry<Integer, Course> course : courses.entrySet()) {
        if (course.getKey() == id) {
            return course.getValue();
        }
    }
    return null;
}
```

- APIServer: Actually creates the server. Notice the setResourceClasses and Provider calls.

```
package it.sapienza.softeng.rest;

import org.apache.cxf.endpoint.Server;
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.lifecycle.SingletonResourceProvider;
```

```java
public class APIServer {

    public static void main(String args[]) throws Exception {
        JAXRSServerFactoryBean factoryBean = new JAXRSServerFactoryBean();
        factoryBean.setResourceClasses(CoursesRepository.class);
        factoryBean.setResourceProvider(new SingletonResourceProvider(new CoursesRepository()));
        factoryBean.setAddress("http://0.0.0.0:8080/");
        Server server = factoryBean.create();

        System.out.println("Server ready...");
        //server.destroy();
        //System.exit(0);
        while (true) {}
    }
}
```

Client

- Student: Like on server, this time also overrides the toString() method to provide a user-friendly print.
- Course: Like on server, this time also overrides the toString*() method to provide a user-friendly print.
- Client: defines the base url, creates a client instance, creates the request(s), optionally with specific body using HttpPut.setEntity(XMLstream), set the Content-Type in the header, executes the request (client.execute()), and closes. Notice in the private methods are provided the instructions to receive and unmarshal the responses. Also for some fucking reason there is a resources folder (next to main folder) in which you can put an xml files that the client can access to load thigns to use as put (see createValidStudent() method)

```java
package it.sapienza.softeng.rest.client;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import javax.xml.bind.JAXB;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpDelete;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.methods.HttpPut;
import org.apache.http.entity.InputStreamEntity;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;

public class Client {

    private static final String BASE_URL = "http://localhost:8080/courses/";
    private static CloseableHttpClient client;

    public static void main(String[] args) throws IOException{
    client = HttpClients.createDefault();
    // Example GET
    Course course = getCourse(1);
    System.out.println(course);
    for (int i = 0; i < course.getStudents().size(); i++) {
        Student student = course.getStudents().get(i);
```

```
        System.out.println(student);
    }
    // Example POST/PUT
    course = getCourse(2);
    System.out.println(course);
    for (int i = 0; i < course.getStudents().size(); i++) {
        Student student = course.getStudents().get(i);
        System.out.println(student);
```

```
for (int i = 0; i < course.getStudents().size(); i++) {
    Student student = course.getStudents().get(i);
    System.out.println(student);
}
createValidStudent();
course = getCourse(2);
System.out.println(course);
for (int i = 0; i < course.getStudents().size(); i++) {
    Student student = course.getStudents().get(i);
    System.out.println(student);
}
client.close();
}

  private static Student getStudent(int courseOrder, int studentOrder) throws IOException {
    final URL url = new URL(BASE_URL + courseOrder + "/students/" + studentOrder);
    final InputStream input = url.openStream();
    return JAXB.unmarshal(new InputStreamReader(input), Student.class);
}
  private static Course getCourse(int courseOrder) throws IOException {
    final URL url = new URL(BASE_URL + courseOrder);
    final InputStream input = url.openStream();
    return JAXB.unmarshal(new InputStreamReader(input), Course.class);
}
private static void createValidStudent() throws IOException {
    final HttpPost httpPost = new HttpPost(BASE_URL + "2/students");
    final InputStream resourceStream =  Client.class.getClassLoader().getResourceAsStream("newStudent.xml");
    httpPost.setEntity(new InputStreamEntity(resourceStream));
    httpPost.setHeader("Content-Type", "text/xml");
    final HttpResponse response = client.execute(httpPost);
}
```

JSON Support and Database

Server

- Flights: as before, just the @ is now @JacksonXmlRootElement
- FlightsRepository: add this to support dbms + the declaration "private Connection conn;". The "pos" is an absolute path where the database is. in maven right click on the project, set configuration, customize, run, arguments, ctrl-v.

```
public void setConnection(String pos) {
    try {
        try {
            Class.forName("org.sqlite.JDBC");
        } catch (ClassNotFoundException ex) {
            Logger.getLogger(FligthsRepository.class.getName()).log(Level.SEVERE, null, ex);
        }
        conn
                = DriverManager.getConnection("jdbc:sqlite:"+pos);
    } catch (SQLException ex) {
```

```
            Logger.getLogger(FligthsRepository.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

- FindById and update now uses SQL logic to search the dbms

```
    private Fligth findById(int id) {

        PreparedStatement stat = null;
        Fligth fl = null;
        try {
            stat = conn.prepareStatement("select * from fligth where id = ?");
            stat.setString(1, String.valueOf(id));

        ResultSet rs = stat.executeQuery();
        if (rs.next()) {
            fl = new Fligth();
            fl.setId(Integer.parseInt(rs.getString("id")));
            fl.setName(rs.getString("name"));
            Logger.getLogger(FligthsRepository.class.getName()).log(Level.INFO, "Accessed : " + fl);
        }
        rs.close();
        } catch (SQLException ex) {
            Logger.getLogger(FligthsRepository.class.getName()).log(Level.SEVERE, null, ex);
        }

        /* simple version
        for (Map.Entry<Integer, Fligth> fligth : fligths.entrySet()) {
            if (fligth.getKey() == id) {
                return fligth.getValue();
            }
        }
        */
        return fl;
    }
```

```
    private void update(int fligthId, Fligth fligth){
        PreparedStatement stat = null;
        try {
            stat = conn.prepareStatement("update fligth set name = ? where id = ?");
            stat.setString(1, fligth.getName());
            stat.setString(2, String.valueOf(fligthId));
        int affectedRow = stat.executeUpdate();
        if (affectedRow == 1) {
            Logger.getLogger(FligthsRepository.class.getName()).log(Level.INFO, "Updated : " + fligth);
            return;
        }
        else throw new RuntimeException();
        }
        catch (Exception ex) {
            Logger.getLogger(FligthsRepository.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

- Server:

```
import com.fasterxml.jackson.jaxrs.json.*;
import java.util.*;
import org.apache.cxf.binding.BindingFactoryManager;
import org.apache.cxf.jaxrs.*;
import org.apache.cxf.jaxrs.lifecycle.*;
```

```java
/**
 *
 * @author studente
 */
public class Server {

    public static void main(String args[]) throws Exception {

        JAXRSServerFactoryBean factoryBean = new JAXRSServerFactoryBean();
        factoryBean.setResourceClasses(FligthsRepository.class);
        FligthsRepository fr = new FligthsRepository();
        fr.setConnection(args[0]);
        factoryBean.setResourceProvider(new SingletonResourceProvider(fr));
        factoryBean.setAddress("http://localhost:8080/");

        List<Object> providers = new ArrayList<Object>();
        providers.add(new JacksonJaxbJsonProvider());

        factoryBean.setProviders(providers);

        BindingFactoryManager manager = factoryBean.getBus().getExtension(BindingFactoryManager.class);
        JAXRSBindingFactory restFactory = new JAXRSBindingFactory();
        restFactory.setBus(factoryBean.getBus());
        manager.registerBindingFactory(JAXRSBindingFactory.JAXRS_BINDING_ID, restFactory);

        org.apache.cxf.endpoint.Server server = factoryBean.create();
        System.out.println("Server ready...");
        while (true) {}
    }
}
```

Client

remember pom. Flight class is the same because data transfer object.

- Client: note that after creating the client we crate an objectmapper and open an input stream on the url. the mapper takes care of marshaling.

```java
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.io.OutputStream;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.methods.HttpPut;
import org.apache.http.entity.InputStreamEntity;
import org.apache.http.entity.StringEntity;

public class Client {

    private static final String BASE_URL = "http://localhost:8080/fligths/";
    private static CloseableHttpClient client;

    public static void main(String[] args) throws IOException {
```

```
        client = HttpClients.createDefault();

        // Example GET
        ObjectMapper mapper = new ObjectMapper();
        URL url = new URL(BASE_URL + "2");
        InputStream input = url.openStream();
        Fligth fl = (Fligth)mapper.readValue(input, Fligth.class);
        System.out.println(fl);

        // Example POST/PUT
        ObjectMapper objectMapper = new ObjectMapper();
        Fligth newFl = new Fligth();
        newFl.setId(4);
        newFl.setName("XX000");
        String json = objectMapper.writeValueAsString(newFl);
        HttpPut httpPut = new HttpPut(BASE_URL + "2/");
        StringEntity entity = new StringEntity(json);
        HttpPut httpPut = new HttpPut(BASE_URL + "2/");
        StringEntity entity = new StringEntity(json);
        httpPut.setEntity(entity);
        httpPut.setHeader("Accept", "application/json");
        httpPut.setHeader("Content-type", "application/json");
        HttpResponse response = client.execute(httpPut);
        System.out.println(response);
        InputStream input2 = url.openStream();
        fl = (Fligth) mapper.readValue(input2, Fligth.class);
        System.out.println(fl);
    }
}
```

Database

Again access configuration panel. This time add new configs. creating: in the arguments put the folder where we want the db to be and the word create. showing has the same path but keyword show, that prints the status of the dbms.

```
public class DBManager {

    public static void main(String[] args) throws Exception {

        Class.forName("org.sqlite.JDBC");
        Connection conn
                = DriverManager.getConnection("jdbc:sqlite:"+args[0]);
        Statement stat = conn.createStatement();

        if (args[1].equals("create")) {
            stat.executeUpdate("drop table if exists fligth;");
            stat.executeUpdate("create table fligth (id, name);");
            PreparedStatement prep = conn.prepareStatement(
                    "insert into fligth values (?, ?);");
            prep.setString(1, "1");
            prep.setString(2, "AZ140");
```

```
            prep.addBatch();
            conn.setAutoCommit(false);
            prep.executeBatch();
            conn.setAutoCommit(true);
            //ETC..
        } else {
            ResultSet rs = stat.executeQuery("select * from fligth;");
            while (rs.next()) {
                System.out.print("Fligth = " + rs.getString("id") + " is : ");
                System.out.println(rs.getString("name"));
            }
            rs.close();
        }
        conn.close();
    }
}
```

fuck this stupid app