School #gradle

Complex Persistence, JAX-RS RESTful, Mockito, Transactions, Rest Client

Óbuda University, Java Enterprise Edition
John von Neumann Faculty of Informatics
Lab 5

Dávid Bedők
2018-01-17
v1.3

Planning and creating a RESTful webservice is a very popular activity nowadays, so we may not miss a REST API from a new enterprise application.

- ▷ compared with the Remote EJB it is a very big jump for the elasticity of the remote communication
- ▷ each reauest is tranfered over HTTP(s) as an HTTP request. This technique is based and uses the structure parts of the HTTP request and response (HTTP method, uri, header, payload, response code, etc.).
- ▷ this is a **webservice**, so cross-platform and the client and server side development would not need to be in one hand
- ▷ this is not as type-safe as the Remote EJB calls (the applied *libraries* will help us to handle the text content in a *type-safe* way)
- ▷ compared to the *SOAP webservice* it can be built faster but it is not as general as the SOAP (a SOAP webservice always has a WSDL document which define almost everything accurately)

RESTful webservices

Planning

A well-designed REST webservice has to be understandable and usable **without documentation** in most cases (or it can be describe itself). Because of that - in my opinion - we can not be used for general purposes (or rather not force it under all circumstances).

Structure

[HTTP-METHOD] `http(s)://{host}:{port}/`
`{context}/{rest-application}/{service}/{operation}`

- ▷ **context**: webapplication context root
  - In the previously presented way we can configure this via the `application.xml` with the help of the applied build system.
- ▷ **rest-application**: root of the REST application (may be empty)
  - it can be configured via the `@ApplicationPath` annotation
- ▷ **service**: root of the coherent business services (may be empty)
  - it can be configured via the `@Path` annotation of the RESTful service class
- ▷ **operation**: path of the RESTful webservice (may be empty)
  - it can be configured via the `@Path` annotation of the RESTful service method

The standard does not forbid to define multiple *rest-application* inside a single EAR, but not all of the application servers support it. We have to consider that during the planning (let the `{context}/{rest-application}/` part of the URI be identical).

Java API for RESTful WebServices
JAX-RS

- ▷ It is part of the Java EE 6 since v1.1
- ▷ JSR 311: JAX-RS
  - https://www.jcp.org/en/jsr/detail?id=311
  - `javax.ws.rs:jsr311-api:1.1`
- ▷ JSR 339: JAX-RS 2.0
  - `javax.ws.rs:javax.ws.rs-api:2.0.1`
  - https://www.jcp.org/en/jsr/detail?id=339
    - 2.2 *"This specification is targeted for Java SE 6.0 or higher and Java EE 6 or higher platforms."*
    - 2.3 *"Additionally, Java EE 6 products will be allowed to implement JAX-RS 2.0 instead of JAX-RS 1.1."*
  - The `javax:javaee-api:6.0` has already contained (but official only the v1.1 is supported)
- ▷ Representational State Transfer (**REST**) architecture
- ▷ Some implementations:
  - Oracle Jersey (RI, Reference Implementation)
  - JBoss RESTeasy
    - `org.jboss.resteasy:resteasy-jaxrs:2.3.10.Final` (latest 2.x)
    - `org.jboss.resteasy:resteasy-jaxb-provider:2.3.10.Final`
  - Apache CXF
- ▷ Its 'pair' library is the **Java API for XML Web Services** (**JAX-WS**) which handles SOAP WebServices, later we are going to learn that. The origin of the JAX-RS abbrevation comes from that.

School

Handle grades of a school

Task: create an Enterprise Java application which can be store and maintain any grades of the student in relation to many different subject.

- ▷ The **student**s are identified by a unique *neptun code*, and beside that we also store his/her *name*s and **institute**s (e.g.: `BANKI`, `KANDO`, `NEUMANN`).
- ▷ Let the **subject**s have unique *name*s, **teacher**s (*name* and *neptun code*) and *description*s.
- ▷ Store a *note* and an exact *timestamp* for each **grade**.

Technology

school project

- We are going to use PostgreSQL RDBMS via JPA. We will introduce the **relations between entities**.
- Amoung some special queries the *addition and deletion of a record* will be presented in details.
- During creation of a **RESTful service** we will learn the basic of **JAX-RS** both server- and client side.
- At the end of the tast we will create **unit tests** in the EJB service layer (*TestNG*, *Mockito*).
- Finally we will touch the opportunities of the **remote debug**.

Build the following RESTful service layer over the realized data tier:

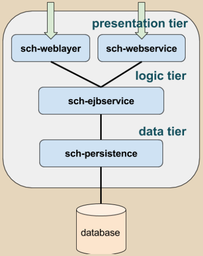
- ▷ **GET** `http://localhost:8080/school/api/student/WI53085`
  - Get the data of the student which neptun code is `WI53085`.
- ▷ **GET** `http://localhost:8080/school/api/student/list`
  - Get all student data.
- ▷ **POST** `http://localhost:8080/school/api/mark/stat`
  - Payload: `Sybase PowerBuilder`
  - For a given subject it generates an average-grade statistics by institutes and by years.
- ▷ **PUT** `http://localhost:8080/school/api/mark/add`
  - Payload: `{"subject": "Sybase PowerBuilder","neptun": "WI53085","grade": "WEAK","note": "Lorem ipsum"}`
  - It saves a new grade in the system.
- ▷ **DELETE** `http://localhost:8080/school/api/student/WI53085`
  - If the student (which neptun code is `WI53085`) has not got any grades, this service will delete that entity from the system.

Project stucture

Subprojects, modules

In case of Mavan there is a **sch-ear** project also.

- ▷ **school** (root project)
  - **sch-webservice** (EAR web module)
    - Project of the RESTful webservices (presentation-tier).
  - **sch-weblayer** (EAR web module)
    - `StudentPingServlet`
    - It contains only a test servlet (presentation-tier).
  - **sch-ejbservice** (EAR ejb module)
    - Business methods (service-tier)
  - **sch-persistence** (EAR ejb module)
    - ORM layer, JPA (data-tier)
  - **sch-restclient** (standalone)
    - Type-safe Java REST client application



There are no requirement for *Remote EJB* calls so the `sch-ejbservice` will stay in one. Both the `sch-webservice` and the `sch-weblayer` project use *Local EJB* calls to reach the `sch-ejbservice` layer.

Database side

School project

`[gradle|maven]\jboss\school\database`

Tables:

- ▷ **institute**
- ▷ **student** (FK: `student_institute_id`)
- ▷ **teacher**
- ▷ **subject** (FK: `subject_teacher_id`)
- ▷ **mark** (FK: `mark_student_id`, `mark_subject_id`)

Relations:

- ▷ 1-N: institute-student
- ▷ 1-N: teacher-subject
- ▷ N-M: student-subject

Persistence layer
School project

Entities:

- ▷ **Mark** (table: `mark`)
- ▷ **Student** (table: `student`)
- ▷ **Subject** (table: `subject`)
- ▷ **Teacher** (table: `teacher`)

Enumeration types:

- ▷ **Institute** (table: `institute`)

EJB Services:

- ▷ `MarkService`
- ▷ `StudentService`
- ▷ `SubjectService`

Subject-Teacher relation

1 subject has got exactly 1 teacher

```java
package hu.qwaevisz.school.persistence.entity;
[..]
@Entity
@Table(name = "subject")
public class Subject implements Serializable {
  [..]
  @ManyToOne(fetch = FetchType.EAGER, optional = false)
  @JoinColumn(name = "subject_teacher_id", referencedColumnName =
      "teacher_id", nullable = false)
  private Teacher teacher;
  [..]
}
```

The `@JoinColumn` annotation describes the relation in the database, so the values in it are table related.

Subject.java

FetchType

- **EAGER**: during the retrieval of the entity the `teacher` relation will be attached automatically (even if there is no direct reference), so the linked data will be availble (e.g. the neptun code of the teacher) (this is the default in case of `@ManyToOne` and `@OneToOne`)
- **LAZY**: the relation will not be attached automatically only if the query asks it or it get a direct reference while the entity is in *attached* state (more effective but requires careful consideration) (this is the default in case of `@OneToMany` and `@ManyToMany`)

Student-Mark relation

1 student has several grades

```
package hu.qwaevisz.school.persistence.entity;
[..]
@Entity
@Table(name = "student")
public class Student imple
  [..]
  @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL,
      mappedBy = "student")
  private final Set<Mark> marks;
  [..]
}
```

The `@OneToMany` and the `@ManyToOne` annotations belong to the ORM model, the referenced fields are the name of the entities' fields (e.g.: `student` instead of `mark_student_id`).

Student.java

One of the most important (and most difficult) thing is configuring the **EAGER** and **LAZY** relations properly. If there are oppsoite claims we can create multiple entities for the same table and the involved relation is EAGER in one and LAZY in the other (in that case LAZY is a more general solution). Using the `@OneToMany` annotation is not mandatory. We use it only if we would like to bind these data from the source entity.
We can use `List<>` and non-generic types as well like `Set/List` interfaces. In the latter case we will need to set the `targetEntity=Mark.class` attribute inside `@OneToMany`. Using a set is more general, more versatile than an ordered list.

Subject-Mark relation

1 subject has got several grades

```
[..]
public class Subject implements Serializable {
  [..]
  @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL,
      mappedBy = "subject")
  private final Set<Mark> marks;
  [..]
  public Subject() {
    this.marks = new HashSet<>
  }
  [..]
}
```

It is a business question to decide: we need to bind the grades for a subject in the ORM, or not. So this relation is optional. Listing the students' grade is more common. You should initialize the collections.

Subject.java

CascadeType

The value of the cascade is a set of CascadeType enums (in case of ALL we do not need to list all of them). These items define which *entity manager* operations will be considered cascading. E.g.: cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}. The default is an empty set.

Mark's relations

Student-Subject N-M relation table

```java
package hu.qwaevisz.school.persistence.entity;
[..]
@Entity
@Table(name = "mark")
public class Mark implements Serializable {
  [..]
  @ManyToOne(fetch = FetchType.EAGER, optional = false)
  @JoinColumn(name = "mark_student_id", referencedColumnName =
       "student_id", nullable = false)
  private Student student;

  @ManyToOne(fetch = FetchType.EAGER, optional = false)
  @JoinColumn(name = "mark_subject_id", referencedColumnName =
       "subject_id", nullable = false)
  private Subject subject;
  [..]
  @Temporal(TemporalType.TIMESTAMP)
  @Column(name = "mark_date", nullable = false)
  private Date date;
  [..]
}
```

The Date should store time, date and both of them at the same time. The @Temporal annotation controls this.

Mark.java

Cascade

Cascading only makes sense only for **parent–child associations** (the parent entity state transition being cascaded to its children entities). Cascading from child to parent is not very useful and usually, it's a mapping *code smell* (it may not intentionally listed in the source code).

CascadeType.PERSIST

We only have to persist the parent entity and all the associated children entities are persisted as well.

CascadeType.DELETE

When the parent entity is deleted, the associated children entities are deleted as well (it is enough to delete the parent only).

Summary of associations

- `@OneToOne`
  - **Task** (`task_id`) → **TaskDetail** (`taskdetail_task_id`)
  - *Task* has exactly one *TaskDetail*
  - use `cascade = CascadeType.ALL` only in the *Task* entity
  - useful the `orphanRemoval = true` attribute as well (in case of *false* when you delete/rewrite (e.g.: batch update) the *Task* id the FK field of *TaskDetail* will become *null*)
- `@OneToMany` és `@ManyToOne`
  - **Task** (`task_id`) → **SubTask** (`subtask_task_id`)
  - *Task* may have several *SubTask*s
  - use the `@OneToMany` in the *Task* and the `@ManyToOne` annotation in the *SubTask* class
- `@ManyToMany`
  - **Bank** (`bank_id`)→ **Account** (`account_bank_id`, `account_client_id`) ← **Client** (`client_id`)
  - *Bank* may have several *Client*s and a *Client* may have several *Account*s in different *Bank*s (but the *Account*s cannot contain more fields (e.g.: accountnumber...))
  - Do not use `CascadeType.ALL` in that case (`PERSIST` + `MERGE` could be enough)
  - in that case the *Account* will not be an entity, a `@JoinTable` annotation will be used instead

Hibernate best practice

Do not use exotic association mappings

Practical test cases for real **many-to-many associations are rare**. Most of the time you need additional information stored in the link table. In this case, it is much better to use two one-to-many associations to an intermediate link class.

In fact, **most associations are one-to-many and many-to-one** (in database level). For this reason, you should proceed cautiously when using any other association style.

Source: http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch26.html

Web application of the REST webservices

sch-webservice subproject

```
apply plugin: 'war'

war { archiveName webserviceArchiveName }

dependencies {
  providedCompile project(':sch-ejbservice')
  // providedCompile group: 'javax.servlet', name: 'javax.servlet-api',
      version: servletapiVersion
  // providedCompile group: 'javax.ws.rs', name: 'javax.ws.rs-api',
      version: jaxrsVersion
  // providedCompile group: 'javax.ws.rs', name: 'jsr311-api', version:
      '1.1'
  providedCompile group: 'javax', name: 'javaee-api', version:
      jeeVersion
}
```

In the *Root* project during the composition of the `application.xml` we set **school** as the context root of this *WEB module*.

build.gradle

Variables of the `root` project:

- ▷ `webserviceArchiveName = 'sch-webservice.war'`
- ▷ `jaxrsVersion = '2.0.1'`

We have more options to set the dependency, one of the simplest way using the *Java EE 6.0 API*. In case of the *JSR311 API* we have to override the `getClasses()` method of the `SchoolRestApplication` class (`return null;`).

REST Application

child of the Application class

```java
package hu.qwaevisz.school.webservice.main;

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

@ApplicationPath("/api")
public class SchoolRestApplication extends Application {

  // @Override
  // public Set<Class<?>> getClasses() {
  // return null;
  // }
}
```

With the help of the @ApplicationPath annotation we can set **api** as the URI of the REST application.

Overrideing the getClasses() method is only necessary if you use the *JSR311 API* dependency.

SchoolRestApplication.java

Student REST service

```java
package hu.qwaevisz.school.webservice;
[..]
@Path("/student")
public interface StudentRestService {
  [..]

  @GET
  @Path("/list")
  @Produces(MediaType.APPLICATION_JSON)
  List<StudentStub> getAllStudents() throws AdaptorException;

  [..]
}
```

With the help of the @Path annotation we are able to set **student** as the URI of the REST service.

Likewise we can use the @Path annotation to set the URI of the REST operation, e.g. this will be **list** in the example.

StudentRestService.java

If we read together the URI parts we will get the following:
http://localhost:8080/school/api/student/list

HTTP Method

The **HTTP Methods** are very important part of the REST services' behavior. Very common usage that many REST API calls differ in the HTTP Method only to entirely support CRUD operations.

- `@POST` $\rightarrow$ Create
- `@GET` $\rightarrow$ Read
- `@PUT` $\rightarrow$ Update
- `@DELETE` $\rightarrow$ Delete
- `@HEAD`
- `@OPTIONS`

Parameter passing in REST operations

- ▷ - (not marked with annotation)
  - The data must be sent to the HTTP Request payload/body element.
- ▷ `@QueryParam("ipsum")`
  - `/lorem?ipsum=42&dolor=sit`
- ▷ `@PathParam("ipsum")`
  - `/lorem/42/xyz`
  - In that case we have to use the `@Path("/lorem/ipsum/xyz")` annotation.
- ▷ `@HeaderParam("ipsum")`
  - Among the keys of the HTTP Request Header there should be one which name is `ipsum`.
  - Special case when we would like to control the **Content-Type** with the `@Consumes` annotation (the data in the payload is compatible with the given MIME type).
  - Special case when we would like to control the **Accept** with the `@Produces` annotation (the data in the Response is compatible with the given MIME type).
- ▷ `@CookieParam("ipsum")`
  - HTTP Request Cookie (In case of browser it is a comfortable solution but the REST services are not depend on the browsers, so it does not recommend to use cookies in RESTful services)
- ▷ `@FormParam("ipsum")`
  - Typically the user sends an `application/x-www-form-urlencoded` (MIME type) *POST* request to the server.
  - In that case the RESTful service will be strongly bound to a webpage. Do not use it if we do not want this dependence.
- ▷ `@MatrixParam("ipsum")`
  - `/lorem;ipsum=42;dolor=sit`
  - It is similar to the `@QueryParam`, but it's purpose is different. If the key-value pair concerns only a part of the URI we should use that type of argument passing (argument fine-tuning)

Query and Path param

There are several mappers which support the **type-safe** argument processing.

- We can use the `String` type (this is the original format of the content)
- All primitive types except **char** (because of the `String`)
- All wrapper classes of primitive types except **Character**
- Any class with a constructor that accepts a single `String` argument
- Any class with the `static` method named `valueOf(..)` that accepts a single `String` argument (each `enum` meets this rule)
- `List<T>`, `Set<T>` or `SortedSet<T>`, where `T` matches the already listed criteria

Default values

The parameter passing does not obligatory (but the calls should not be ambiguous). If a parameter has not got a value than in case of primitive the value will be the *default* (zero literal), in case of collection it will be an empty `List`/`Set` or `SortedSet`, and any other cases it will be `null`. You can use a `@DefaultValue` annotation as well, and we can redefine the default values.

Process forms

Sample code

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
  [..]
}
```

For form prcessing a simple `Servlet` is enough in most cases.

Free processing of an URL

@Context annotation

Entirely free processing of a given URL:

```
@GET
public String get(@Context UriInfo ui) {
  MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
  MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

Free processing of the HTTP Header:

```
@GET
public String get(@Context HttpHeaders hh) {
  MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
  Map<String, Cookie> pathParams = hh.getCookies();
}
```

We can use in a very similar way the `HttpServletRequest` and `HttpServletContext` instances, and all of them can be injected into the implementation class as well (so we do not need to use these in the interface):

```
public class Sample {

  @Context
  private HttpHeaders headers;

  @Context
  private HttpServletRequest servletRequest;
  [..]
}
```

HTTP Response

Response builder

The return value of the REST method will be the HTTP Response's payload according to the adjusted MIME type. If we use the `Response` return type we will have much more options to configure the HTTP Response (but the interface will not be *type-safe* enough).

```java
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;
[..]
Response.ok().build(); // 200 OK
Response.noContent().build(); // 204 No Content
Response.status(Status.NOT_FOUND).entity([..])
  .type(MediaType.APPLICATION_JSON).build();
```

Get student data

GET `http://localhost:8080/school/api/student/{neptun}`

Stub vs. Entity

So far the Stubs and the Entities were almost the same, but this is not necessarily the case. The *customer* needs (stubs) may contain some elements which e.g.:

- ▷ Redundants: for obvious reasons we will not store redudant information on database (and in the level of entities).
- ▷ A type of the field is different in the level of entities and stubs (may be common covering the details of a type and use a `String` type instead in the level of the stubs, but in that case we weaken the *type-safe* behavior).
- ▷ We use localized constants in the level of the stubs, or we apply some kind of business defined names (the localization might come from the client side only, it is entirely independent from the database).
- ▷ A stub may contain some information which are independent from the related entity, e.g. these information come from an other system. From the client's point of view this separation (the data comes from two systems) are invisible and irrelevant.

Get student's data

GET `http://localhost:8080/school/api/student/WI53085`

We would like to show some kind of depth walk result of the `Student`.

```
{
    "name": "Juanita A. Jenkins",
    "neptun": "WI53085",
    "institute": "BANKI",
    "marks": [
        {
            "subject": {
                "name": "Sybase PowerBuilder",
                "teacher": {
                    "name": "Richard B. Cambra",
                    "neptun": "UT84113"
                },
                "description": "Donec rhoncus lacus quis est cursus aliquet."
            },
            "grade": "WEAK",
            "note": "Lorem ipsum",
            "date": 1477902214713,
            "gradeValue": 2
        },
        [..]
    ],
    "numberOfMarks": 3
}
```

The `numberOfMarks` and the `gradeValue` are computed fields.

RESTful Endpoint (sch-webservice project)

```java
@Path("/student")
public interface StudentRestService {

  @GET
  @Path("/{neptun}")
  @Produces(MediaType.APPLICATION_JSON)
  StudentStub getStudent(@PathParam("neptun") String neptun)
      throws AdaptorException;

  [..]
}
```

StudentRestService.java

Get student data

Top-Down approach

```
@GET
@Path("/{neptun}")
@Produces(MediaType.APPLICATION_JSON)
StudentStub getStudent(@PathParam("neptun")
     String neptun) throws AdaptorException;
```

- ▷ `sch-webservice`
  - `StudentRestService`
  - `StudentRestServiceBean` SLSB
    -
    - To be able to inject EJB into the class, the EJB context must be seen. One of the option for that if we make an SLSB (you can also use CDI).
- ▷ `sch-ejbservice`
  - `StudentFacade` Local interface
    - `StudentStub getStudent(String neptun) throws AdaptorException;`
  - `StudentRestServiceBean` SLSB
- ▷ `sch-persistence`
  - `StudentService` Local interface
    - `Student read(String neptun) throws PersistenceServiceException;`
  - `StudentServiceImpl` SLSB
  - `Student` entity
- ▷ `sch-ejbservice`
  - `StudentConverter` Local interface
  - `StudentConverterImpl` SLSB
    - Create the *accessor* methods of the computed fields

```
SELECT st
FROM Student st
  LEFT JOIN FETCH st.marks m
  LEFT JOIN FETCH m.subject su
  LEFT JOIN FETCH su.teacher
WHERE st.neptun=:neptun
```

```
[..]
public class StudentStub {
  [..]
  public int getNumberOfMarks() {
    return this.marks.size();
  }
  [..]
}
```

Generated native queries

```sql
SELECT
  student0_.student_id AS student_1_2_0_,
  marks1_.mark_id AS mark_id1_0_1_,
  subject2_.subject_id AS subject_1_3_2_,
  teacher3_.teacher_id AS teacher_1_4_3_,
  student0_.student_institute_id AS student_2_2_0_,
  student0_.student_name AS student_3_2_0_,
  student0_.student_neptun AS student_4_2_0_,
  marks1_.mark_date AS mark_dat2_0_1_,
  marks1_.mark_grade AS mark_gra3_0_1_,
  marks1_.mark_note AS mark_not4_0_1_,
  marks1_.mark_student_id AS mark_stu5_0_1_,
  marks1_.mark_subject_id AS mark_sub6_0_1_,
  marks1_.mark_student_id AS mark_stu5_2_0__,
  marks1_.mark_id AS mark_id1_0_0__,
  subject2_.subject_description AS subject_2_3_2_,
  subject2_.subject_name AS subject_3_3_2_,
  subject2_.subject_teacher_id AS subject_4_3_2_,
  teacher3_.teacher_name AS teacher_2_4_3_,
  teacher3_.teacher_neptun AS teacher_3_4_3_
FROM
  student student0_
    LEFT OUTER JOIN mark marks1_ ON
      student0_.student_id=marks1_.mark_student_id
    LEFT OUTER JOIN subject subject2_ ON
      marks1_.mark_subject_id=subject2_.subject_id
    LEFT OUTER JOIN teacher teacher3_ ON
      subject2_.subject_teacher_id=teacher3_.teacher_id
WHERE
  student0_.student_neptun=?
```

The **FETCH** inside the *JPQL* query will get (and fill) the children entities' data (SELECT block).

The **LEFT JOIN** of the *JPQL* query will wire the children entites (FROM block). The LEFT join is required because it may occur that a student has not got any grade, and without this the student data would not be fetched too.

Get all students' data

GET `http://localhost:8080/school/api/student/list`

RESTful Endpoint (sch-webservice project)

```
@Path("/student")
public interface StudentRestService {

  @GET
  @Path("/list")
  @Produces(MediaType.APPLICATION_JSON)
  List<StudentStub> getAllStudent() throws AdaptorException;

  [..]
}
```

StudentRestService.java

Fetch all students' data

The entire operation can be done if we reuse the previous *named query* (of course we have to take out the neptun filtering from the `WHERE` block). But right now we are going to present a **typical bad example**, a quite simple *JPQL* query and analyze the realized events/queries:

```
SELECT s
FROM Student s
ORDER BY s.name
```

Result: `org.hibernate.LazyInitializationException` at `StudentConverterImpl` class. The entity which we read from the database and send to the *facade layer* becomes **detached**, the *entity manager* cannot perform any operations on it, it cannot supervise it. When the converter service tries to call the `getMarks()` method of the `Student`, the container 'notices' that if it simply give back a `null`, it causes a false/uncertain state of the system (we did not fetch it, se we do not know that the student has grades or not). This can only occur with `LAZY` fetchType.

What could be the solution (workaround)?

- Rewrite the fetchType to **EAGER**: probably this is the most comfortable solution and it will work immediately. Only one 'little' issue here: the JPA will generate **10 native queries** to create the expected data (the number depends how many different subject/teacher are affected). And do not forget that we modify the entity so any other queries could be affected.
- Read some **LAZY** references (grades) on the entity during its **attached** state. With this we ask the *entity manager* to ensure these data for us: it will cause **10 additional queries** again, but this time we will not influence other operations at least.

```
[..]
public class StudentServiceImpl implements StudentService {
  [..]
  @Override
  public List<Student> readAll() throws PersistenceServiceException {
    [..]
    result = this.entityManager.createNamedQuery(Student.GET_ALL,
         Student.class).getResultList();
    for (final Student student : result) {
      student.getMarks().size();
    }
    [..]
  }
  [..]
}
```

Implement paging

In any case, is JOIN FETCH the ultimate solution?

In case of a list every time comes up the claim that we would not want to fetch all the data, only N items (pageSize) from a K offset (page). In native queries there are the LIMIT and the OFFSET keywords (it could be database dependent). In JPA we can achieve the same on the TypedQuery/Query instance:

```java
List<Student> result =
    this.entityManager.createNamedQuery(Student.GET_ALL,
    Student.class).setFirstResult((page - 1) *
    pageSize).setMaxResults(pageSize).getResultList();
```

Attention!

It will not put the LIMIT keyword into the native query automatically, but it will work in all case. How can this be? If an entity fetch a child entity as well, the 'first K rows' most likely will not be the expected K rows (so we cannot use the LIMIT keyword and the ORM knows it). The first 'K' rows probably will contain the first main entity and some of its children entities (in RDBMS the main entity's columns will be repeated). That is why JPA will fetch all data and filter the records afterwards. In case of big tables it causes **serious performance and resource problems**, and often the developer does not even know about it.

RESTful Endpoint (sch-webservice project)

```java
@Path("/student")
public interface StudentRestService {

  @GET
  @Path("/list/{page}")
  @Produces(MediaType.APPLICATION_JSON)
  Response getStudents(@DefaultValue("3") @QueryParam("pagesize")
      int pageSize, @PathParam("page") int page) throws
      AdaptorException;

  [..]
}
```

StudentRestService.java

Effective implementation of paging

GET `http://localhost:8080/school/api/student/list/2?pagesize=5`

```java
public class StudentServiceImpl implements StudentService {
  [..]
  @Override
  public List<Student> read(int pageSize, int page) throws PersistenceServiceException {
    if (LOGGER.isDebugEnabled()) {
      LOGGER.debug("Get Students (pageSize: " + pageSize + ", page: " + page + ")");
    }
    List<Student> result = null;
    try {
      result = this.entityManager.createNamedQuery(Student.GET_ALL,
           Student.class).setFirstResult((page - 1) * pageSize).setMaxResults(pageSize)
          .getResultList();
      List<Long> studentIds =
           result.stream().map(Student::getId).collect(Collectors.toList());
      result = this.entityManager.createNamedQuery(Student.GET_BY_IDS,
           Student.class).setParameter("ids", studentIds).getResultList();
    } catch (final Exception e) {
      throw new PersistenceServiceException("Unknown error when fetching Students! " +
           e.getLocalizedMessage(), e);
    }
    return result;
  }
  [..]
}
```

```sql
SELECT s
FROM Student s
ORDER BY s.name
```

```sql
SELECT st
FROM Student st
  LEFT JOIN FETCH st.marks m
  LEFT JOIN FETCH m.subject su
  LEFT JOIN FETCH su.teacher
WHERE st.id IN :ids
```

Generated queries

```
SELECT
  student0_.student_id AS student_1_2_,
  student0_.student_institute_id AS student_2_2_,
  student0_.student_name AS student_3_2_,
  student0_.student_neptun AS student_4_2_
FROM
  student student0_
ORDER BY
  student0_.student_name
LIMIT ?
OFFSET ?
```

```
SELECT
  student0_.student_id AS student_1_2_0_,
  marks1_.mark_id AS mark_id1_0_1_,
  subject2_.subject_id AS subject_1_3_2_,
  [..]
  subject2_.subject_teacher_id AS subject_4_3_2_,
  teacher3_.teacher_name AS teacher_2_4_3_,
  teacher3_.teacher_neptun AS teacher_3_4_3_
FROM
  student student0_
    LEFT OUTER JOIN mark marks1_
      ON student0_.student_id=marks1_.mark_student_id
    LEFT OUTER JOIN subject subject2_
      ON marks1_.mark_subject_id=subject2_.subject_id
    LEFT OUTER JOIN teacher teacher3_
      ON subject2_.subject_teacher_id=teacher3_.teacher_id
WHERE
  student0_.student_id IN ( ? , ? , ? , ? )
```

Average grade statistics

POST `http://localhost:8080/school/api/mark/stat`

Postman

Testing REST API

- https://www.getpostman.com/
- Version : **5.3.2**
- Free for individual users but there is a Pro version which supports team work
- We can test a `GET` request with a simple browser, but in more complicated cases we have to create small (X)HTML pages, and this could be very cumbersome and difficult to maintain.
- For automatic tests there will be some scripts or source codes (it is not a major challenge), but for ad-hoc testing, supporting the development, a ready-to-use solution is always expedient. The **Postman** is such a tool like this.
- The projects can be syncronized with Google account.

Average grade statistics

POST `http://localhost:8080/school/api/mark/stat`

Based on a given subject (*payload*) we want to create an average grade statistics (*average*) which is grouped by institute (*group-by*) and year (*group-by*).

HTTP Request payload (text):

```
Sybase PowerBuilder
```

HTTP Response (application/json):

```
[
    {
        "institute": "KANDO",
        "year": 2012,
        "averageGrade": 4
    },
    {
        "institute": "KANDO",
        "year": 2013,
        "averageGrade": 4
    },
    {
        "institute": "NEUMANN",
        "year": 2014,
        "averageGrade": 3.5
    }
]
```

RESTful Endpoint (sch-webservice project)

```java
@Path("/mark")
public interface MarkRestService {

	@POST
	@Path("/stat")
	@Produces("application/json")
	List<MarkDetailStub> getMarkDetails(String subject) throws
	    AdaptorException;

	[..]
}
```

MarkRestService.java

Is there any issue here?!

- ▷ We have to group the grades by years, but we do not have such data in that form. Of course we have the information in the timestamp field and we are able to get the necessary data with a *PostgreSQL* function (`DATE_TRUNC('year', mark_date)` or `EXTRACT('year' FROM mark_date)`). But in JPA there is a few issues:
  - There are various date functions in Hibernate and in Eclipselink, we can use these in HQL/EQL queries, but these have not got standard form (or the support is inadequate) (this issues is not related to the date functions: you will face that issue when you have to use e.g. a custom DB function in JPQL).
  - There is solution to register DB functions in JPA but this is also implementation dependent (JPA 2.0 supports DB function calls without supervision).
- ▷ At a given complexity point maintaining a query in JPQL can be cumbersome
  - It is a fairy tale that somebody can create a query in JPQL but cannot do the same in ANSI SQL. Always (always!) we build the query in ANSI SQL first thereafter in JPQL (this may happen in head, but without this you cannot create optimum queries in JPQL).
  - The complex queries represent value in the product, these are part of the source code (language inside a language), and the storage and the maintanance are important → here comes (into view) the database side **VIEW**s.

Not always it is worth "to force" the purely Java/ORM solution. In the spirit of simplicity, dare to disassemble the responsibility between ORM and RDBMS.

Native query

```
SELECT
  markdetail.student_institute_id,
  markdetail.mark_year,
  AVG(markdetail.mark_grade)
FROM
  (
    SELECT
      mark_subject_id,
      student_institute_id,
      mark_grade,
      DATE_PART('year', mark_date) AS mark_year
    FROM mark
      INNER JOIN student ON ( mark_student_id = student_id )
    WHERE ( 1 = 1 )
  ) AS markdetail
WHERE ( 1 = 1 )
  AND ( markdetail.mark_subject_id = 2 )
GROUP BY
  markdetail.student_institute_id,
  markdetail.mark_year
ORDER BY
  markdetail.student_institute_id,
  markdetail.mark_year
```

The subject will be given by name. In the example the aggreagation (join) of the subject table is missing.

Solution plan

Requirements:

- ▷ have to create a **group-by** query by **institute** and **year**
- ▷ have to **prefilter** the data by **subject**

Database VIEW creation:

- ▷ We produce the field where we have to use the database function (e.g.: `DATE_PART`).
- ▷ we have to include all the fields to which one of the following is true:
  - to which we have to prefilter the data (`subject_id`)
  - to which we have to group the results (`institute_id` and `mark_year` (computed field))
  - to which we have to use in the aggregation function (e.g. `AVG`) later (`mark_grade`)

Attention!

It is very rare that a database VIEW contains a group-by query, because we cannot perform any additional filtering later.

Database VIEW

```sql
CREATE VIEW markdetail AS
  SELECT
    ROW_NUMBER() OVER() AS markdetail_id,
    mark_subject_id AS markdetail_subject_id,
    student_institute_id AS markdetail_institute_id,
    mark_grade AS markdetail_grade,
    DATE_PART('year', mark_date) AS markdetail_year
  FROM mark
    INNER JOIN student ON ( mark_student_id = student_id )
  WHERE ( 1 = 1 );
```

VIREW becomes an entity in the ORM level, and each entities must have a primary key. The `ROW_NUMBER()` suitables for it (we will not update or delete any rows of the *view*, moreover we will use that VIEW in a group-by query, so the individual rows are unimportant).

VIEW testing

We should create the following queries in ORM

```
SELECT
  markdetail_institute_id,
  markdetail_year,
  AVG(markdetail_grade)
FROM
  markdetail
    INNER JOIN subject ON
      ( markdetail_subject_id = subject_id )
WHERE ( 1 = 1 )
  AND ( subject_name = 'Sybase PowerBuilder' )
GROUP BY
  markdetail_institute_id,
  markdetail_year
ORDER BY
  markdetail_institute_id,
  markdetail_year;
```

VIEW in the ORM layer

Each fields are handled like in a normal entity. Do not use different programming 'rules' between VIEW and TABLE. Bind the `subject` and the `institute` fields like in other entities.

```
@Entity
@Table(name = "markdetail")
public class MarkDetail implements Serializable {

  @Id
  @Column(name = "markdetail_id", nullable = false)
  private Long id;

  @ManyToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL, optional = false)
  @JoinColumn(name = "markdetail_subject_id", referencedColumnName = "subject_id",
       nullable = false)
  private Subject subject;

  @Enumerated(EnumType.ORDINAL)
  @Column(name = "markdetail_institute_id", nullable = false)
  private Institute institute;

  @Column(name = "markdetail_grade", nullable = false)
  private Integer grade;

  @Column(name = "markdetail_year")
  private Integer year;

  [..]
}
```

MarkDetail.java

JPQL and generated native queries

```
SELECT new hu.qwaevisz.school.persistence.result.MarkDetailResult(
  md.institute,
  md.year,
  AVG(md.grade) )
FROM MarkDetail md
WHERE md.subject.name=:subject
GROUP BY md.institute, md.year
ORDER BY md.institute, md.year
```

The result of the **JPQL** query is a type/set, and each field/items in this type/set is an *institute*, a *year* and an *real average grade* value. We do not have an entity like that in the ORM layer, so we will create a **result** type for that purpose (`MarkDetailResult`).

```
SELECT
  markdetail0_.markdetail_institute_id AS col_0_0_,
  markdetail0_.markdetail_year AS col_1_0_,
  AVG(markdetail0_.markdetail_grade) AS col_2_0_
FROM
  markdetail markdetail0_ CROSS JOIN subject subject1_
WHERE
  markdetail0_.markdetail_subject_id=subject1_.subject_id
  AND subject1_.subject_name=?
GROUP BY
  markdetail0_.markdetail_institute_id ,
  markdetail0_.markdetail_year
ORDER BY
  markdetail0_.markdetail_institute_id,
  markdetail0_.markdetail_year
```

MarkDetailResult

```java
package hu.qwaevisz.school.persistence.result;
[..]
public class MarkDetailResult {

  private final Institute institute;

  private final Integer year;

  private final double averageGrade;

  public MarkDetailResult(Institute institute, Integer year,
      double averageGrade) {
    this.institute = institute;
    this.year = year;
    this.averageGrade = averageGrade;
  }

  [..]
}
```

This class is not an entity, just a **simple DTO**. The constructor is useful for business reasons, do not need to create *default* ctor-s (this is required in case of entities).

MarkDetailResult.java

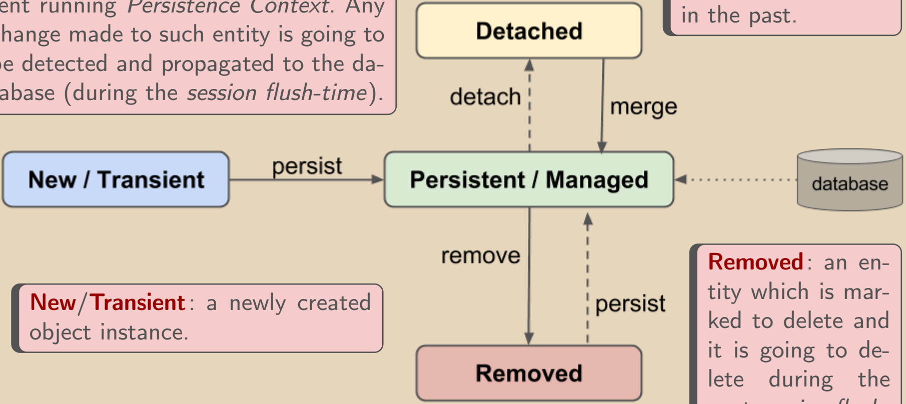
Add new grade

PUT `http://localhost:8080/school/api/mark/add`

JPA - Entity states

Persistent/Managed: the entity has been associated with a database table row and it's being managed by the current running *Persistence Context*. Any change made to such entity is going to be detected and propagated to the database (during the *session flush-time*).

Detached: an entity which was *managed* in the past.



New/Transient: a newly created object instance.

Removed: an entity which is marked to delete and it is going to delete during the next *session flush-time*.

Add new grade

PUT `http://localhost:8080/school/api/mark/add`

HTTP Request payload (application/json):

```
{
  "subject": "Sybase PowerBuilder",
  "neptun": "WI53085",
  "grade": "WEAK",
  "note": "Lorem ipsum"
}
```

The grade in the request is a business-defined constant (`WEAK`) which is unknown for the persistence layer.

HTTP Response (application/json):

```
{
  "subject": {
    "name": "Sybase PowerBuilder",
    "teacher": {
      "name": "Richard B. Cambra",
      "neptun": "UT84113"
    },
    "description": "Donec"
  },
  "grade": 2,
  "note": "Lorem ipsum",
  "date": 1443797867042
}
```

RESTful Endpoint (sch-webservice project)

```java
@Path("/mark")
public interface MarkRestService {

  @PUT
  @Path("/add")
  @Consumes("application/json")
  @Produces("application/json")
  MarkStub addMark(MarkInputStub stub) throws AdaptorException;

  [..]
}
```

MarkRestService.java

Problems arising from the lack of transaction management

So far we learned how to create queries and we could miss the transaction handling with ease. But during data manipulation we cannot postpone further.

- ▷ The subject that belongs a grade which the user would like to create just now is deleted in an other transaction at the 'same time', or it is renamed to something else.
- ▷ The student is deleted from the system in a parallel transaction.

The transaction handling is much more important in some other business related aspect as well:

- ▷ It can be imagine that the system acknowledges the grade creation, but when the user retrieves the data of the student the entity does not exist anymore. Both transactions were executed successfully, from the point of view of the 'program' everything is good, but the user will not be satisfied, (s)he would expect some notification.
- ▷ The user could retrieve the data of the user but when (s)he would like to create a new grade the system send him/her a message that the student does not exist. The state of the application is not inconsistent (because of the normalized database schema), our user will not be satisfied even so.

Transaction Management

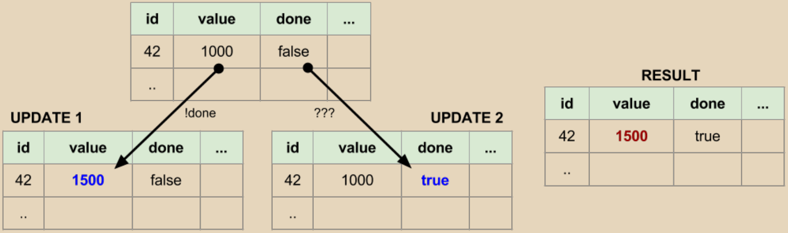
Validation and rollback situation

In the first round the importance of the transaction management is determining **which operations** (queries + data manipulations) **have to execute in the same transaction**:

- e.g.: before inserting a new grade we have to verify in the same transaction that the student and the subject are exist and we can notify the user why we cannot perform the recording (otherwise the `INSERT` will fail and the we have *mine* the possible reason from the returned exception (e.g. foreign key constrained failed, etc.)). *If the validation is a meaningful business use case we will do it before the data manipulation.*
- We have to insert multiple records into different tables (e.g.: one row into a *parent* table and `N` rows into its *child* table). If - even the last - one single insert is failed in any reason, none of the rows may stay in the database (**rollback** situation). The ORM layer joins these operations better then the RDBMS, but regardless of that this situation is valid in the ORM as well.

Successful operations in case of parallel executions

There are some happening where each transaction is performed successfully still **it will be formed a business wrong situatation**, because the two separate transactions do not mutually prevent each other.
One of the most common way when the same record are modified parallel, e.g. one of the *actor* would like to change the *amount to be paid* field (`value`) meanwhile an other *actor* would like to fulfill/approve the transfer of this amount and updates a flag (`done`) to true.



id	value	done	...
42	1000	false	
..			

UPDATE 1

id	value	done	...
42	**1500**	false	
..			

UPDATE 2

id	value	done	...
42	1000	**true**	
..			

RESULT

id	value	done	...
42	**1500**	true	
..			

From one side we can protect the `UPDATE 1` operation with a validation (`!done`), but we cannot do that in the other direction. For such situations the **locking** will be the solution.

Lock strategies

Pessimistic Locking

Lock the database table row for the time of the transaction which is initiated the locking. Until the transaction do not notify the termination, any other incomming request will be queued. It may cause performance drops with ease, it could be the *bottleneck* of the system). You have to be careful not to develop **deadlock** situation (two (or more) transactions lock 1-1 records separately, none of them release it while each transaction would like to use the other transaction's locked record, so both of them will wait).

Optimistic Locking

In the previous mentioned example the issue of the `UPDATE 2` is that it cannot able to perform a validation/check before executes the `done` flag update to true. For the *actor* point of view it would be important to see exactly the same record which (s)he is going to be approved. One of the solution could be generate the *hash* of the columns and before the *update* we compare the sent and the recalculated hashes. More effecient solution if we use a *version number* or a *timestamp* for the same purpose (but in these cases we need an additional column in the table). If this validation is failed, we say that the record is *dirty* and the transaction will be rollbacked. In case of database pool, this strategy is more than recommended.

XA Datasource

eXtended Architecture

The standard of the ***D**istributed **T**ransaction **P**rocessing* (DTP) which describes the interface between the global and local *transaction manager*. Solving the ACID[1] operations is its essential purpose (e.g.: it can commit/rollback transactions amoung multiple database, or between a database and a message queue). The implementation of XA is mostly based on **two-phase commit**.

Two-phase commit (2PC)

One of the type of the **A**tomic **C**ommitment **P**rotocol (ACP). The transactions must be represent as an atomic item (it cannot be separated). Its name comes from the following: each operation stands for a **voting** and a **completion** phase. In the first phase the related components must be signal back to the 'coordinator' (the component is ready to perform the operation (the resource is free, available, the network is fine, etc.) or not). When all the related components' votes were 'true', the coordinator asks the components the actual execution in the second phase. The components will signal back the result again (commit/rollback).

[1] **A**tomicity, **C**onsistency, **I**solation, **D**urability

Transaction attributes

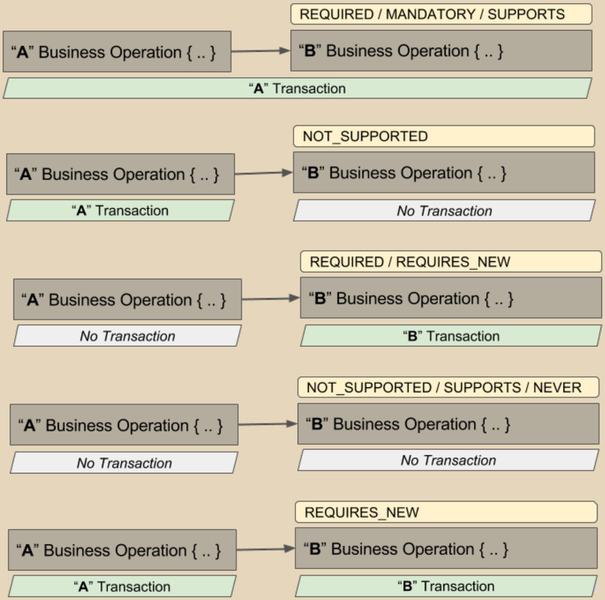
`@TransactionAttribute` annotation

If we call an EJB service through proxy from a Servlet, an **EJB transaction** may be started. We can configure this EJB transaction via the `@TransactionAttribute` annotation which we put onto the business method[2] or the containing class.
You can only use it if the *container* handles the transactions (this is the default behavior, but if we want we can put the
`@TransactionManagement(TransactionManagementType.CONTAINER)` annotation onto the bean).
The client side calls the 'remote' business service. The annotation appears always on the 'remote' business method:

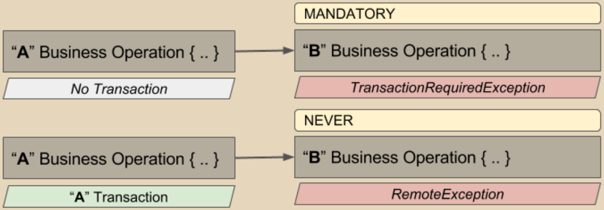
- **MANDATORY**: If a client invokes the enterprise bean's method while the client is associated with a transaction context, the container invokes the enterprise bean's method in the client's transaction context (must use the transaction of the client).
- **NEVER**: The client is required to call without a transaction context, otherwise an exception is thrown.
- **NOT_SUPPORTED**: The container invokes an enterprise bean method whose transaction attribute NOT_SUPPORTED with an unspecified transaction context (do not need transactions, may improve performance).
- **REQUIRED** (default): If a client invokes the enterprise bean's method while the client is associated with a transaction context, the container invokes the enterprise bean's method in the client's transaction context.
- **REQUIRES_NEW**: The container must invoke an enterprise bean method whose transaction attribute is set to REQUIRES_NEW with a new transaction context.
- **SUPPORTS**: If the client calls with a transaction context, the container performs the same steps as described in the REQUIRED case (you should use the Supports attribute with caution).

[2] in case of *session bean* or *message driven bean*

Transaction attributes - Successful cases


REQUIRED / MANDATORY / SUPPORTS
"A" Business Operation { .. }
"B" Business Operation { .. }
"A" Transaction
NOT_SUPPORTED
"A" Business Operation { .. }
"B" Business Operation { .. }
"A" Transaction
No Transaction
REQUIRED / REQUIRES_NEW
"A" Business Operation { .. }
"B" Business Operation { .. }
No Transaction
"B" Transaction
NOT_SUPPORTED / SUPPORTS / NEVER
"A" Business Operation { .. }
"B" Business Operation { .. }
No Transaction
No Transaction
REQUIRES_NEW
"A" Business Operation { .. }
"B" Business Operation { .. }
"A" Transaction
"B" Transaction

Transaction attributes - Unsuccessful cases


MANDATORY
"A" Business Operation { .. }
"B" Business Operation { .. }
No Transaction
TransactionRequiredException
NEVER
"A" Business Operation { .. }
"B" Business Operation { .. }
"A" Transaction
RemoteException

MarkFacadeImpl SLSB (sch-ejbservice project)

```java
package hu.qwaevisz.school.ejbservice.facade;
[..]
@Stateless(mappedName = "ejb/markFacade")
public class MarkFacadeImpl implements MarkFac

  @EJB
  private StudentService studentService;
  @EJB
  private SubjectService subjectService;
  @EJB
  private MarkService markService;
  @EJB
  private MarkConverter converter;

  @Override
  @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
  public MarkStub addMark(String subject, String neptun, int grade, String note) throws
       AdaptorException {
    try {
      final Long subjectId = this.subjectService.read(subject).getId();
      final Long studentId = this.studentService.read(neptun).getId();
      return this.converter.to(this.markService.create(studentId, subjectId, grade,
           note));
    } catch (final PersistenceServiceException e) {
      LOGGER.error(e, e);
      throw new AdaptorException(ApplicationError.UNEXPECTED, e.getLocalizedMessage());
    }
  }
  [..]
}
```

Because of the `REQUIRES_NEW` annotation this business function will run in an independent transaction (if we call this from a *Servlet* this will be the result too (def. `REQUIRED`)). Each business methods are annotated with `REQUIRED` which we use inside that method.

MarkFacadeImpl.java

MarkServiceImpl SLSB (sch-persistent project)

```java
package hu.qwaevisz.school.persistence.service
[..]
@Stateless(mappedName = "ejb/markService")
@TransactionManagement(TransactionManagementTy
public class MarkServiceImpl implements MarkSe

  @PersistenceContext(unitName = "sch-persiste
  private EntityManager entityManager;

  @Override
  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public Mark create(final Long studentId, final Long subjectId, final Integer grade,
       final String note) throws PersistenceServiceException {
    try {
      final Student student = this.entityManager.find(Student.class, studentId);
      final Subject subject = this.entityManager.find(Subject.class, subjectId);
      Mark mark = new Mark(student, subject, grade, note);
      this.entityManager.persist(mark);
      this.entityManager.flush();
      return mark;
    } catch (final Exception e) {
      throw new PersistenceServiceException("..." + e.getLocalizedMessage(), e);
    }
  }
  [..]
}
```

Only **attached** (managed) entity can be saved (persist or merge). Knowing the ID we can easily 'create' attached entities with the `find()` operation of the *entity manager* (inside transaction it will not generate new queries in this case).

You have to pay attention **not to detach the same entity twice**. You will get 'Multiple representations of the same entity are being merged.' exception in that case and for instance you have to remove the `CascadeType.MERGE`/`CascadeType.PERSIST` flag in some relations.

Generated native queries

```
SELECT
  subject0_.subject_id AS subject_1_3_,
  [..]
  subject0_.subject_teacher_id AS
       subject_4_3_
FROM subject subject0_
WHERE s

SELECT
  teach
  teach
       teacher_2_4_0_,
  teacher0_.teacher_neptun AS
       teacher_3_4_0_
FROM teacher teacher0_
WHERE teacher0_.teacher_id=?
```

Validation of the **Subject**: **2 SELECT** (`Subject.teacher EAGER`).

```
SELECT
  student0_.student_id AS
       student_1_2_0_,
  marks1_.ma
  [..]
  teacher3_.
       teach
FROM
  student student0_
    LEFT OUTER JOIN mark marks1_ ON
    student0_.student_id=marks1_.mark_studen
    LEFT OUTER JOIN subject subject2_ ON
    marks1_.mark_subject_id=subject2_.subjec
    LEFT OUTER JOIN teacher teacher3_ ON
    subject2_.subject_teacher_id=teacher3_.t
WHERE student0_.student_neptun=?
```

Validation of the **Student**: **1 SELECT** (optimized).

```
SELECT
  NEXTVAL ('mark_mark_id_seq')

INSERT INTO mark
  (mark_date, mark_grade, mark_note, mark_student_id,
      mark_subject_id, mark_id)
VALUES
  (?, ?, ?, ?, ?, ?)
```

Insertion of **Mark**: **1 SELECT + 1 INSERT**.

Remove student

DELETE `http://localhost:8080/school/api/student/{neptun}`

Remove student

DELETE `http://localhost:8080/school/api/student/{neptun}`

`http://localhost:8080/school/api/student/ABC123`

Response status code: **400 Bad Request**

```
{
  "code": 40,
  "message": "Resource not found",
  "fields": "ABC123"
}
```

`http://localhost:8080/school/api/student/WI53085`

Response status code: **412 Precondition Failed**

```
{
  "code": 50,
  "message": "Has dependency",
  "fields": "WI53085"
}
```

`http://localhost:8080/school/api/student/TX78476`

Response status code: **204 No Content**

RESTful Endpoint (sch-webservice project)

```
@Path("/student")
public interface StudentRestService {

  @DELETE
  @Path("/{neptun}")
  void removeStudent(@PathParam("neptun") String neptun) throws
      AdaptorException;

  [..]
}
```

StudentRestService.java

HTTP (response) status codes

http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Successful 2xx

- ▷ 200 OK
- ▷ 201 Created
- ▷ 202 Accepted
- ▷ 204 No Content
- ▷ 206 Partial Content

Redirection 3xx

- ▷ 300 Multiple Choices
- ▷ 301 Moved Permanently
- ▷ 302 Found
- ▷ 303 See Other
- ▷ 304 Not Modified
- ▷ 307 Temporary Redirect

Client Error 4xx

- ▷ 400 Bad Request
- ▷ 401 Unauthorized
- ▷ 402 Payment Required
- ▷ 403 Forbidden
- ▷ 404 Not Found
- ▷ 405 Method Not Allowed
- ▷ 408 Request Timeout
- ▷ 412 Precondition Failed
- ▷ 413 Request Entity Too Large
- ▷ 414 Request-URI Too Long
- ▷ 415 Unsupported Media Type

Server Error 5xx

- ▷ 500 Internal Server Error
- ▷ 501 Not Implemented
- ▷ 503 Service Unavailable

Error handling on RESTful interface

- Based on the *HTTP Response status code* we may send back different 'payloads', because all receiving clients can easily differentiate that situation. In case of error an `ErrorStub` instance will contain the business error code in `JSON` format, possibly some other **public information** as well. There are not exist standard solution like the SOAP Fault.
- In an object-oriented application the error cases of a business methods are exceptions. We use this 'special' return values to differentiate the failures. In common when a business error occurs we throws a *checked* exception (instance of `AdaptorException`) which contains some **proected data** (these are help us to detect/prevent the error later) beside the public information. In `JAX-RS` we can create an `ExceptionMapper<T>` class which transforms the business error into an *HTTP Response* (`AdaptorExceptionMapper`).

It follows from the above that the `AdaptorException` will be the factory of the `ErrorStub`. It would be redundant if we always add all `ErrorStub` fields when an error occurs (it would be difficult to maintain if e.g. we have to change a business error code). Avoid the above we are going to define an `ApplicationError` *enum* which encapsulates the redundant parts of the `ErrorStub`. Thereby in case of an error we just need to give an instance of this enum and the of course the changing part(s) (e.g. fields).

ErrorStub (sch-ejbservice project)

```java
package hu.qwaevisz.school.ejbservice.domain;

public class ErrorStub {

  private int code;
  private String message;
  private String fields;

  public ErrorStub(int code, String message, String fields) {
    this.code = code;
    this.message = message;
    this.fields = fields;
  }

  [..]
}
```

ErrorStub.java

ApplicationError (sch-ejbservice project)

```java
package hu.qwaevisz.school.ejbservice.util;
import javax.ws.rs.core.Response.Status;
import hu.qwaevisz.school.ejbservice.domain.ErrorStub;
public enum ApplicationError {

  UNEXPECTED(10, Status.INTERNAL_SERVER_ERROR, "Unexpected error"),
  NOT_EXISTS(40, Status.BAD_REQUEST, "Resource not found"),
  HAS_DEPENDENCY(50, Status.PRECONDITION_FAILED, "Has dependency");

  private final int code;
  private final Status httpStatus;
  private final String message;

  private ApplicationError(int code, Status httpStatus, String message) {
    this.code = code;
    this.httpStatus = httpStatus;
    this.message = message;
  }

  public Status getHttpStatus() {
    return this.httpStatus;
  }
  public int getHttpStatusCode() {
    return this.httpStatus.getStatusCode();
  }
  public ErrorStub build(String field) {
    return new ErrorStub(this.code, this.message, field);
  }
}
```

The `ApplicationError` enum instance is the factory of the `ErrorStub`, and it is able le get (and store) the HTTP Status value which is also correlated the type of the error.

ApplicationError.java

AdaptorException (sch-ejbservice project)

```java
package hu.qwaevisz.school.ejbservice.exception;
import hu.qwaevisz.school.ejbservice.domain.ErrorStub;
import hu.qwaevisz.school.ejbservice.util.ApplicationError;
public class AdaptorException extends Exception {

  private final ApplicationError error;
  private final String fields;

  public AdaptorException(ApplicationError error, String message,
      String fields) {
    this(error, message, null, fields);
  }
  [..]
  public Status getHttpStatus() {
    return this.error.getHttpStatus();
  }

  public ErrorStub build() {
    return this.error.build(this.fields);
  }
}
```

The AdaptorException instance is the factory of ErrorStub (it delegates the task to the instance of ApplicationError enum).

AdaptorException.java

AdaptorExceptionMapper (sch-webservice project)

```java
package hu.qwaevisz.school.webservice.mapper;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;
import hu.qwaevisz.school.ejbservice.exception.AdaptorException;

@Provider
public class AdaptorExceptionMapper implements
    ExceptionMapper<AdaptorException> {

  @Override
  public Response toResponse(final AdaptorException e) {
    return Response.status(e.getHttpStatus())
      .entity(e.build())
      .type(MediaType.APPLICATION_JSON)
      .build();
  }

}
```

The @Provider classes offer configuration possibilities for the JAX-RS. The implementation of this API already has several providers (e.g.: object-XML/JSON two-way conversion).

AdaptorExceptionMapper.java

StudentFacadeImpl (sch-ejbservice project)

```java
public class StudentFacadeImpl implements StudentFacade {
  @EJB
  private StudentService studentService;
  @EJB
  private MarkService markService;

  @Override
  @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
  public void removeStudent(final String neptun) throws AdaptorException {
    try {
      if (this.studentService.exists(neptun)) {
        if (this.markService.count(neptun) == 0) {
          this.studentService.delete(neptun);
        } else {
          throw new AdaptorException(ApplicationError.HAS_DEPENDENCY, "Student has
               undeleted mark(s)", neptun);
        }
      } else {
        throw new AdaptorException(ApplicationError.NOT_EXISTS, "Student doesn't
             exist", neptun);
      }
    } catch (final PersistenceServiceException e) {
      LOGGER.error(e, e);
      throw new AdaptorException(ApplicationError.UNEXPECTED, e.getLocalizedMessage());
    }
  }
}
```

The transaction attributes of all the three persistent operations is REQUIRED, so the queries and the delete will be executed in the same transaction.

StudentFacadeImpl.java

StudentServiceImpl (sch-persistence project)

```java
public class StudentServiceImpl implements StudentService {
  @PersistenceContext(unitName = "sch-persistence-unit")
  private EntityManager entityManager;

  @Override
  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void delete(final String neptun) throws
      PersistenceServiceException {
    if (LOGGER.isDebugEnabled()) {
      LOGGER.debug("Remove Student by neptun (" + neptun + ")");
    }
    try {
      this.entityManager.createNamedQuery(Student.REMOVE_BY_NEPTUN).
        neptun).executeUpdate();
    } catch (final Exception e) {
      throw new PersistenceServiceException("Unknown error when
        removing Student by neptun (" + neptun + ")! " +
        e.getLocalizedMessage(), e);
    }
  }
}
```

StudentServiceImpl.java

JPQL queries

Does the student exist?

```
SELECT COUNT(s)
FROM Student s
WHERE s.neptun=:neptun
```

If it exists, does (s)he got any grades?

```
SELECT COUNT(m)
FROM Mark m
WHERE m.student.neptun=:neptun
```

If not then perform the deletion:

```
DELETE FROM Student s
WHERE s.neptun=:neptun
```

Cross-Origin Resource Sharing (CORS)

CORS is a technique that the browsers (*user agent*) ask permission to send (and receive) HTTP requests from a service which locates in an *other* domain (*other* means that it differs from the originally called domain).
In such case the browser sends an `OPTION` HTTP request to the server (with the header (and url) data of the original request) to ask the permission. After that **the server side component's responsibility processes this `OPTION` request and decides that the requester's IP address (client) may call the service or not**. Very common solution that the CORS filter at the server side allows HTTP requests from everywhere. We are going to do that too. If the server denies the request, the *user agent* will not send the original request.

There are 3rd party solutions for the CORS filters, but we are not going to use these right now.

CORS - Process OPTION requests

```java
@Path("/student")
public interface StudentRestService {
  [..]

  @OPTIONS
  @Path("{path:.*}")
  Response optionsAll(@PathParam("path") String path);
}
```

StudentRestService.java

```java
public class StudentRestServiceBean implements StudentRestService
    {
  [..]
  @Override
  public Response optionsAll(final String path) {
    return Response.status(Response.Status.NO_CONTENT).build();
  }
}
```

StudentRestServiceBean.java

CORS Filter

```java
package hu.qwaevisz.school.webservice.filter;
[..]
@WebFilter(filterName = "SchoolCrossOriginResourceSharingFilter", urlPatterns = { "/*" 
    })
public class SchoolCORSFilter implements Filter {

  public static final String ALLOW_ORIGIN = "Access-Control-Allow-Origin";
  public static final String ALLOW_CREDENTIALS = "Access-Control-Allow-Credentials";
  public static final String ALLOW_METHODS = "Access-Control-Allow-Methods";
  public static final String ALLOW_HEADERS = "Access-Control-Allow-Headers";
  public static final String MAX_AGE = "Access-Control-Max-Age";

  @Override
  public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
        FilterChain chain)
      throws IOException, ServletException {
    final HttpServletResponse response = (HttpServletResponse) servletResponse;
    response.setHeader(ALLOW_ORIGIN, "*");
    response.setHeader(ALLOW_METHODS, "GET, POST, PUT, DELETE, OPTIONS, HEAD");
    response.setHeader(MAX_AGE, "1209600");
    response.setHeader(ALLOW_HEADERS, "x-requested-with, origin, content-type, accept,
        X-Codingpedia, authorization");
    response.setHeader(ALLOW_CREDENTIALS, "true");
    response.setHeader("Cache-Control", "no-cache");
    chain.doFilter(servletRequest, servletResponse);
  }
  [..]
}
```

SchoolCORSFilter.java

JBoss debug

Remote JVM debug options

```
> [JBOSS_HOME]/bin/standalone.[bat|sh] --debug
> [JBOSS_HOME]/bin/standalone.[bat|sh] --debug [DEBUG-PORT]
```

Default debug port: **8787**

```
[..]
Listening for transport dt_socket at address: 8787
[..]
```

server.log

You can remote debug any JVM (if you have permission to reach that process), you only need to add the following arguments to the java starter command (the -Xdebug is the older version but the newer JVMs recognize it):

```
-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=[DEBUG-PORT]
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=[DEBUG-PORT]
```

WebLogic debug

Remote JVM debug options

[WL-HOME] | user_projects | domains | mydomain | startWebLogic.cmd

```
set JAVA_OPTIONS=-Xdebug -Xnoagent
    -Xrunjdwp:transport=dt_socket,address=4000,server=y,suspend=n
```

startWebLogic.cmd

Debug port: **4000**

- -XDebug → enable debug
- -Xnoagent → turn off the default sun.tools.debug debug *agent*
- -Xrunjdwp → load the a JDWP[3]'s JPDA[4] reference implementation

[3] Java™ Debug Wire Protocol

[4] Java™ Platform Debugger Architecture

Eclipse debug

Debug remote JVM

Run | Debug Configurations |

▷ Remote Java Application

- Local menu : New
- Project : Browse.. (otherwise this is irrelevant)
- Connection Type : Standard (Socket Attach)
  - Host : **localhost**
  - Port : **8787**
- Apply and Debug

▷ Switch to Debug Perspective

- In the Debug view we have to see the threads
- At the same place, local menu : Edit source lookup
  - Add Java Project(s)

Creating unit tests

Unit testing

Dávid Bedők: **Programozási feladatok megoldási módszertana** (Óbudai Egyetem, 2015, Hungarian)
Chapter 5.2: Unit testing

`http://users.nik.uni-obuda.hu/bedok.david/progi-felok-megoldasi-moda-latest.pdf`

The purpose of the unit test creation is not scope of this lab. The **TestNG** 3rd party library was introduced earlier (in a very superficial way). The focus is now on the unit testing of the **EJB services**. How do we test classes which resources (the proxies of other EJB/CDI beans) were injected by a container or framework at runtime? We are going to get to know the **Mockito** 3rd party library.

Mockito

Mocking/Faking 3rd party library

http://site.mockito.org/
Version: **2.12** (2017.11)
Artifact: `org.mockito:mockito-core:2.12.0`

Why should I use mock techniques?

Firstly this is important because if you use the real implementation of the other class and this class has a bug, more than one test cases will failed because of the same issue.
Secondly there are exceptions where some cases you will use the real implementation and not a mock of the class. But you always have to give reasons in these situations.

Unit testing of average grade statistics

sch-ejbservice project

```
List<MarkDetailStub> getMarkDetails(String subject) throws
    AdaptorException;
```

What are the responsibilities of that method in that layer (whitebox testing[5])?

- ▷ With a given subject's name the method produces the list of `MarkDetailStub` output.
- ▷ With the subject's name the method asks the statistics data from the persistence layer (list of `MarkDetailResult`)
- ▷ Ask the transformation/conversion of the list from the persistence layer to get the list of `MarkDetailStub` which can be interpretable by the user
- ▷ When something goes wrong in the persistence layer the method has to notify the unexpected event (`ApplicationError.UNEXPECTED`).
- ▷ If the subject does not exist, the method should return an empty list.

There is no any other responsibilities for that business method (e.g.: it does not care the details of the conversion or which *named query* should be performed in the persistence layer (there is a database side VIEW or not), etc.).

[5] we know the internal operation details

MarkFacadeImplTest (`sch-ejbservice` project)

Test setup

```
package hu.qwaevisz.school.ejbservice.facade;
[..]
public class MarkFacadeImplTest {

  @InjectMocks
  private MarkFacadeImpl facade;

  @Mock
  private MarkService markService;

  @Mock
  private MarkConverter markConverter;

  @BeforeMethod
  public void setup() {
    MockitoAnnotations.initMocks(this);
  }

  [..]
}
```

The tested class has the `@InjectMocks` annotation. The framework will inject the mocks into this class. **Do not instantiate** the `MarkFacadeImpl` instance!

The classes which have the `@Mock` annotation will be the mock objects. These mocks will be injected into the tested class. The mocks have to be prepared before usage mostly (in contrast the fakes will be created as a ready-to-use instances).

To call the `MockitoAnnotataions.initMocks(this)` method is very important. This method does the injection (we can put this line into a parent class).

`MarkFacadeImplTest.java`

MarkFacadeImplTest (sch-ejbservice project)

There are no statistical data

```
public class MarkFacadeImplTest {
  [..]
  private static final String SUBJECT_NAME = "LoremIpsumSubject";

  @Test
  public void returnAnEmptyListWhenTheSubjectIsNotExistsOrHasntGotAnyGrades() throws
       AdaptorException, PersistenceServiceException {
    final List<MarkDetailResult> results = new ArrayList<>();
    Mockito.when(this.markService.read(SUBJECT_NAME)).thenReturn(results);
    final List<MarkDetailStub> stubs = new ArrayList<>();
    Mockito.when(this.markConverter.to(results)).thenReturn(stubs);

    final List<MarkDetailStub> markDetailStubs =
         this.facade.getMarkDetails(SUBJECT_NAME);

    Assert.assertEquals(markDetailStubs.size(), 0);
  }
  [..]
}
```

MarkFacadeImplTest.java

It have to be a requirement that the `markService.read()` method returns an empty list if there are no grades for the subject, or the subject is not even exist. The test verifies that the `getMarkDetails()` method does not failed in that case.

MarkFacadeImplTest (sch-ejbservice project)

Successful case

```java
public class MarkFacadeImplTest {
  @Test
  public void createListOfMarkDetailsFromSubjectName() throws AdaptorException,
       PersistenceServiceException {
    final List<MarkDetailResult> results = new ArrayList<>();
    results.add(new MarkDetailResult(Institute.NEUMANN, 2000, 0));
    results.add(new MarkDetailResult(Institute.KANDO, 2000, 0));
    Mockito.when(this.markService.read(SUBJECT_NAME)).thenReturn(results);
    final List<MarkDetailStub> stubs = new ArrayList<>();
    final MarkDetailStub neumannStub = Mockito.mock(MarkDetailStub.class);
    stubs.add(neumannStub);
    final int yearKando = 2014;
    final double averageGradeKando = 2.4142;
    stubs.add(new MarkDetailStub(Institute.KANDO.toString(), yearKando,
         averageGradeKando));
    Mockito.when(this.markConverter.to(results)).thenReturn(stubs);

    final List<MarkDetailStub> markDetailStubs =
         this.facade.getMarkDetails(SUBJECT_NAME);

    Assert.assertEquals(markDetailStubs.size(), 2);
    Assert.assertEquals(markDetailStubs.get(0), neumannStub);
    Assert.assertEquals(markDetailStubs.get(1).getInstitute(),
         Institute.KANDO.toString());
    Assert.assertEquals(markDetailStubs.get(1).getYear(), yearKando);
    Assert.assertEquals(markDetailStubs.get(1).getAverageGrade(), averageGradeKando);
  }
}
```

MarkFacadeImplTest.java

MarkFacadeImplTest (`sch-ejbservice` project)

Unsuccessful case

```java
public class MarkFacadeImplTest {
  [..]

  @Test(expectedExceptions = AdaptorException.class)
  public void
       throwUnexpectedApplicarionErrorIfSomethingErrorOccoursInThePersistenceLayer()
       throws PersistenceServiceException, AdaptorException {
    Mockito.when(this.markService.read(SUBJECT_NAME)).thenThrow(PersistenceServiceExceptio
    this.facade.getMarkDetails(SUBJECT_NAME);
    Assert.fail();
  }

  [..]
}
```

`MarkFacadeImplTest.java`

One of the field (`ApplicationError` enum) of the thrown `AdaptorException` contains `javax.ws.rs.core.Response.Status` instances. That is why we have to add the e.g. the `org.jboss.spec:jboss-javaee-6.0` artifact to the test *classpath* (in Gradle we may use the `compileOnly` *dependency configuration* instead of the `compile` in case of the JavaEE API).

Typical scenario

```
Subject subject = Mockito.mock(Subject.class);
```

It creates a `Subject` mock (the `@Mock` annotation creates a mock like this too, but with the annotation the library injects the mock into the tested class, if we ask it and it is possible).

```
Mockito.when(this.markService.read(SUBJECT_NAME))
   .thenReturn(results);
```

It prepares a mock. In this case when the `read()` method is called with the given `String` value, the mock will return a `results` (which is a list of mocks, but it can be real instance/literal as well).

```
Mockito.verify(this.markService).read(SUBJECT_NAME);
```

It verifies the accurate method call of a mock (does the `read()` method is called with the given `String` literal). If the call is missing, the test will be failed.

Mockito additional possibilities

- It can be an option to throw an exception in case of `when()` (and we can do that with the method which has not got return value (`void`)).
- With the help of the `Matchers` we could avoid to set the accurate values, we only need to add e.g. the type of the argument (we can combine the options if the method has more than one arguments).
- We can catch mock's inner arguments (and after that we are able to write an `Assert` with that value).
- We can define how many times we would like to `verify()` a method call.
- We can set multiple return values in order in case of `when()` if the same mock will be called multiple times.
- etc.

Do not overengineering

If you test your class too deep it will be very sensitive for any tiny changes as well and you cannot refactor your production code easily. Because of this try to avoid the usage of the `verify()` (for instance in the presented example these verifies are totally unnecessary).

List of filtered student's grades

POST `http://localhost:8080/school/api/mark/get/{neptun}`

Filtered grades of student

POST `http://localhost:8080/school/api/student/marks/{neptun}`

HTTP Request payload (application/xml):

```
<markcriteria>
  <subject>Programming</subject>
  <minimumgrade>1</minimumgrade>
  <maximumgrade>3</maximumgrade>
</markcriteria>
```

HTTP Response (application/xml):

```
<?xml version="1.0" encoding="UTF-8" standalone="ye
<marks>
    <mark>
        <date>2014-09-29T04:15:34+02:00</date>
        <grade>MEDIUM</grade>
        <gradeValue>3</gradeValue>
        <note>Phasellus</note>
        <subject>
            <description>Fusce [..] purus.</description>
            <name>Python Programming</name>
            <teacher>
                <name>Christine W. Culp</name>
                <neptun>OK73109</neptun>
            </teacher>
        </subject>
    </mark>
    [..]
</marks>
```

With that service we are able to list the student's (`neptun`) grades which meet the given conditions: part/term of the subject's name (`subject`), the lower (`minimumgrade`) and upper (`maximumgrade`) limit of the grade.

RESTful Endpoint (sch-webservice project)

```java
@Path("/student")
public interface StudentRestService {
  [..]
  @POST
  @Consumes("application/xml")
  @Produces("application/xml")
  @Path("/marks/{neptun}")
  @Wrapped(element = "marks")
  List<MarkStub> getMarks(@PathParam("neptun") String neptun,
      MarkCriteria criteria) throws AdaptorException;
  [..]
}
```

StudentRestService.java

@Wrapped

With the help of `org.jboss.resteasy.annotations.providers.jaxb.Wrapped` annotation we can wrap the parent element of the `List<T>` type (in case of XML it is useful, but in JSON it does not). Because of that annotation we have to add the `org.jboss.resteasy:resteasy-jaxb-provider` compile time artifact to the classpath.

JPQL and generated native queries

```
SELECT m
FROM Mark m
  JOIN FETCH m.student
  JOIN FETCH m.subject s
  JOIN FETCH s.teacher
WHERE m.student.neptun=:studentNeptun
  AND m.grade BETWEEN :minGrade AND :maxGrade
  AND m.subject.name LIKE CONCAT('%',:subjectNameTerm,'%')
```

```
SELECT
  mark0_.mark_id as mark_id1_0_0_,
  student1_.student_id as student_1_2_1_,
  subject2_.subject_id as subject_1_3_2_,
  teacher3_.teacher_id as teacher_1_4_3_,
  [..]
  teacher3_.teacher_neptun as teacher_3_4_3_
FROM mark mark0_
  INNER JOIN student student1_ ON mark0_.mark_student_id=student1_.student_id
  INNER JOIN subject subject2_ ON mark0_.mark_subject_id=subject2_.subject_id
  INNER JOIN teacher teacher3_ ON subject2_.subject_teacher_id=teacher3_.teacher_id
WHERE student1_.student_neptun=?
  AND ( mark0_.mark_grade BETWEEN ? AND ? )
  AND ( subject2_.subject_name LIKE ('%'||?||'%') )
```

REST Client application

```java
private static final String REQUEST_PAYLOAD = "" //
    + "<markcriteria>" //
    + "  <subject>Programming</subject>" //
    + "  <minimumgrade>1</minimumgrade>" //
    + "  <maximumgrade>3</maximumgrade>" //
    + "</markcriteria>";
public static void main(String[] args) throws IOException {
  URL url = new URL("http://localhost:8080/school/api/student/marks/WI53085");
  HttpURLConnection connection = (HttpURLConnection) url.openConnection();
  connection.setRequestMethod("POST");
  connection.setRequestProperty("Content-Type", "application/xml");
  connection.setUseCaches(false);
  connection.setDoOutput(true);
  DataOutputStream outputStream = new DataOutputStream(connection.getOutputStream());
  outputStream.writeBytes(REQUEST_PAYLOAD);
  outputStream.close();

  InputStream inputStream = connection.getInputStream();
  BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
  StringBuilder response = new StringBuilder();
  String line;
  while ((line = reader.readLine()) != null) {
    response.append(line);
  }
  reader.close();

  System.out.println(response);
}
```

Calling a RESTful service is entirely language independent, we only need to build HTTP Requests, we can do that 'almost' from any programming languages. The presented Java sample creates a POST request and processes the response. Of course the XML payload is available in plain text format.

REST Client application

type-safe solution

Using the `java.net` package there are some issues: it is not provide us *type-safe* solution and it causes lots of *boilerplate* source codes. Of course there are some 3rd party libraries with are tring to correct these but luckily the popular implementations of JAX-RS are also support the client side operations, so we can use them on the REST client side as well, even in a Java SE application (JBoss **RESTeasy** and Oracle **Jersey**).

Do we use it in couple?

It does not matter which library we are using on the server and client sides. When we use RESTeasy on the server we should use Jersey on the client. It does also not matter which *stub* you are using, the only thing that matters is the content will be serialized based on the adjusted MIME type (the client side should be able to product the appropriate XML/JSON content).

REST Client

sch-restclient project

JAXB

The **JAXB Provider** (**J**ava **A**rchitecture for **X**ML **B**inding) is good for serialization and deserialization of XML(s).

```
jar { archiveName 'sch-restclient.jar' }

dependencies {
  compile group: 'org.jboss.spec', name: 'jboss-javaee-6.0',
      version: jbossjee6Version
  compile group: 'org.jboss.resteasy', name:'resteasy-jaxrs',
      version: resteasyVersion
  compile group: 'org.jboss.resteasy',
      name:'resteasy-jaxb-provider', version: resteasyVersion
  compile group: 'commons-logging', name: 'commons-logging',
      version: commonsloggingVersion
}
```

bui

```
ext {
  jbossjee6Version = '3.0.3.Final'
  resteasyVersion = '2.3.7.Final'
  commonsloggingVersion = '1.2'
}
```

RESTful Remote Endpoint (sch-restclient project)

```java
package hu.qwaevisz.school.restclient;
[..]
@Path("/student")
public interface StudentRemoteRestService {

  @POST
  @Consumes("application/xml")
  @Produces("application/xml")
  @Path("/marks/{student}")
  @Wrapped(element = "marks")
  ClientResponse<List<MarkStub>>
      getFilteredMarks(@PathParam("student") String neptun,
      MarkConditions conditions);
}
```

StudentRemoteRestService.java

The StudentRemoteRestService differs at some points intentionally (for presentation purpose) from the server side StudentRestService (e.g.: in the path we use student key, the name of the MarkCriteria class is something else). Usage of the ClientResponse<T> is practical because we can handle the *header* and the *response code* of the *HTTP Response*, not only the *entity*.

Sample code (sch-ejbservice project)

```java
public List<MarkStub> process(String studentNeptun,
    MarkConditions conditions) {
  URI serviceUri =
      UriBuilder.fromUri("http://localhost:8080/school/api").build()
  ClientRequestFactory crf = new ClientRequestFactory(serviceUri);

  StudentRemoteRestService api =
      crf.createProxy(StudentRemoteRestService.class);
  ClientResponse<List<MarkStub>> response =
      api.getFilteredMarks(studentNeptun, conditions);

  LOGGER.info("Response status: " + response.getStatus());
  MultivaluedMap<String, Object> header = response.getMetadata();
  for (final String key : header.keySet()) {
    LOGGER.info("HEADER - key: " + key + ", value: " +
        header.get(key));
  }
  List<MarkStub> marks = response.getEntity();
  return marks;
}
```

SchoolRestClient.java