School #gradle

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

Óbudai Egyetem, Java Enterprise Edition

Műszaki Informatika szak

Labor 5

Bedők Dávid

2018-01-17

v1.3

A RESTful webservice készítése és tervezése manapság igen divatos, új enterprise alkalmazás tervezéséből sose hagyjuk ki a REST API-t.

- A korábban megismert Remote EJB világához képest egy hatalmas ugrás az alkalmazással való távoli kommunikáció szempontjából
- Minden kérés az HTTP(s)-n keresztül, HTTP Request formájában jut el a serverhez. A megoldás az HTTP kérés és válasz szerkezetei elemeire épít (HTTP method, uri, header, payload, response code, stb.)
- **webszolgáltatás**ról van szó, így cross-platform megoldás, mely során egyáltalán nem követelmény az, hogy a server és kliens oldali fejlesztés egy kézben legyen
- ugyanakkor nem beszélhetünk olyan szintű type-safe viselkedésről sem, mint pl. a Remote EJB esetén (az alkalmazott *library*-k segítségével fogunk a szöveges tartalomból *type-safe* forráskódot kezelni)
- A *SOAP webservice*-zel párba állítva tipikusan egy sokkal gyorsabban építhető, ámbár kevésbé általános webszervíz technológiáról beszélhetünk (egy SOAP webservice-t minden esetben egyértelműen definiál egy WSDL dokumentum)

Egy jól megtervezett REST webservice-nek **dokumentáció nélkül** is egyértelműnek is használhatónak "kell" lennie a legtöbb esetben (avagy önmagát kell tudnia leírni, részleteznie, rávezetnie a használatra). Ennél fogva - saját véleményem szerint - nem használható teljesen általános célra (avagy nem érdemes minden körülmények között erőltetni).

Szerkezet

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

- **context**: webapplication context root
  - A korábban megismert módon, az `application.xml`-en keresztül az alkalmazott build rendszer segítségével definiálhatjuk értékét.
- **rest-application**: REST alkalmazás root-ja (lehet üres)
  - egy `@ApplicationPath` annotáción keresztül állíthatjuk be értékét
- **service**: üzletileg egy csoportba tartozó szolgáltatások root-ja (lehet üres)
  - a RESTful service-t definiáló osztályon, egy `@Path` annotáción keresztül állítjuk értékét
- **operation**: a RESTful webservize hivatkozása (lehet üres)
  - a RESTful service-t definiáló metódusok, egy `@Path` annotáción keresztül állítjuk értékét

A standard nem tiltja egy EAR-on belül több *rest-application* definiálását, azonban ezt nem minden alkalmazás szerver támogatja, így tervezéskor vegyük figyelembe hogy `{context}/{rest-application}/` rész minden üzleti műveletek esetén azonos legyen.

Java API for RESTful WebServices

JAX-RS

- Java EE 6 része v1.1 óta
- 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."*
  - A `javax:javaee-api:6.0` már tartalmazza (azonban elvben 1.1-et támogat a Java EE 6)
- Representational State Transfer (**REST**) architektúra
- Néhány implementáció:
  - 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
- "Párja" a **Java API for XML Web Services** (**JAX-WS**), mely a SOAP WebService-ek kezelésére szolgál, később lesz róla szó. A JAX-RS rövidítés eredete a korábban sokkal népszerűbb SOAP WS-ek rövidítéséből ered.

Feladat: hozzunk létre egy Enterprise Java alkalmazást, mely hallgatók érdemjegyeit adott tantárgy vonatkozásában tárolja és kezeli.

- A **diák**okat *neptun kódjuk* egyedien azonosítja, e mellett tároljuk el *nevüket* és **intézmény**üket (pl.: `BANKI`, `KANDO`, `NEUMANN`).
- A **tantárgy**aknak legyen egy *egyedi nevük*, **oktató**juk (*név* és *neptun kód*) illetve *leírásuk*.
- Minden **érdemjegy**hez tároljunk el egy *megjegyzést* és egy pontos *időbélyeget* is.

- A megvalósítás során PostgreSQL RDBMS adatmodellre épített JPA-n keresztül megszólított ORM réteg kerül építésre. Immáron a fizikai táblák közötti kapcsolat az **entitások közötti kapcsolatok**ban is meg fog mutatkozni.
- Speciális lekérdezések mellett az *új rekord rögzítésének* és *törlésének* mikéntjét is alaposabban megvizsgáljuk.
- A **RESTful service** megvalósítása kapcsán megismerkedünk a **JAX-RS** alapjaival, mind szerver- és kliens oldalon.
- A feladat végén **egység teszteket** fogunk készíteni az EJB service rétegben (*TestNG*, *Mockito*).
- Végül a **Remote debug** lehetőségeire is kitekintünk.

A megvalósított tárolási réteg fölé az alábbi RESTful service réteget építsük fel:

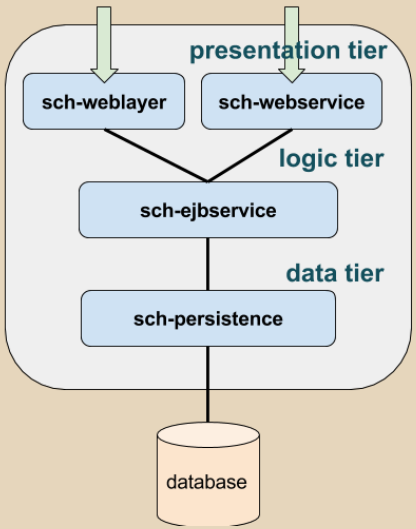
- **GET** `http://localhost:8080/school/api/student/WI53085`
  - A `WI53085` neptun kóddal rendelkező hallgató adatait adja vissza.
- **GET** `http://localhost:8080/school/api/student/list`
  - Az összes hallgató adatait adja vissza.
- **POST** `http://localhost:8080/school/api/mark/stat`
  - Payload: `Sybase PowerBuilder`
  - Egy adott tantárgy vonatkozásában visszaad egy intézményre és évekre bontott átlag-eredmény statisztikát.
- **PUT** `http://localhost:8080/school/api/mark/add`
  - Payload: `{"subject": "Sybase PowerBuilder","neptun": "WI53085","grade": "WEAK","note": "Lorem ipsum"}`
  - Adott érdemjegyet rögzít a rendszerben.
- **DELETE** `http://localhost:8080/school/api/student/WI53085`
  - A `WI53085` neptun kóddal rendelkező hallgatót törli a rendszerből, ha nincsenek rögzített érdemjegyei.

Project stuktúra

Alprojektek, modulok

- **school** (root project)
  - **sch-webservice** (EAR web module)
    - A RESTful webservice-ek project-je (pre
  - **sch-weblayer** (EAR web module)
    - `StudentPingServlet`
    - Kizárólag ezt az egy teszt servletet tartalmazó webproject (presentation-tier).
  - **sch-ejbservice** (EAR ejb module)
    - Üzleti metódusok (service-tier)
  - **sch-persistence** (EAR ejb module)
    - ORM réteg, JPA (data-tier)
  - **sch-restclient** (standalone)
    - Type-safe Java REST kliens alkalmazás

Maven esetén egy **sch-ear** project is megjelenik.



Most nincs követelmény *Remote EJB* hívásokra, így az `sch-ejbservice` egyben marad. Az `sch-webservice` és az `sch-weblayer` is *Local EJB* hívásokkal éri el az `sch-ejbservice` réteget.

Adatbázis oldal

School project

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

Táblák:

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

Kapcsolatok:

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

Persistence réteg

School project

Entity-k:

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

Felsorolás típus:

- **Institute** (tábla: `institute`)

EJB Service-ek:

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

Subject-Teacher reláció

1 tantárgynak pontosan 1 oktatója van

```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;
  [..]
}
```

A `@JoinColumn` annotáció a fizikai adatbázissal való kapcsolatot írja le, a benne szereplő értékek a fizikai táblára vonatkoznak.

Subject.java

FetchType

- **EAGER**: az entitás lekérésekor automatikusan kapcsolja a `teacher` táblát (akkor is ha erre nincs direkt kérés), ezáltal elérhetőek lesznek a kapcsolt adatok (pl. tanár neptun kódja) (`@ManyToOne` és `@OneToOne` esetén alapértelmezett)
- **LAZY**: nem csatolja automatikusan, csak ha erre kéri a lekérdezés vagy *attached* állapotban hivatkoznak a kapcsolatra (hatékonyabb, de körültekintést igényel) (`@OneToMany` és `@ManyToMany` esetén alapértelmezett)

Student-Mark reláció

1 diáknak számos jegye lehet

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

A `@OneToMany` és a `@ManyToOne` annotációk az ORM modellre vonatkozik, a hivatkozott mezők field-ek nevei (pl. `student` és nem `mark_student_id`).

`Student.java`

Az egyik legfontosabb (és legnehezebb) dolog megfelelően beállítani az **EAGER** és **LAZY** kapcsolatokat. Ha ellentétes igények merülnek fel, akkor sincs feltétlenül probléma: ugyanarra a táblára akármennyi entitást készíthetünk, egyikben a kapcsolat lehet EAGER, a másikban LAZY, de ilyen esetben a LAZY lesz az általánosabb megoldás. A `@OneToMany` használata egyáltalán nem kötelező. Csak akkor vegyük fel, ha az adott irányban az adatra szükségünk van üzletileg.
Lehet használni `List<>`-et és nem generikus `Set/List` interface-t is. Utóbbi esetben szükség lesz egy `targetEntity=Mark.class` attribútumra is a `@OneToMany`-n belül. Halmaz használata általánosabb, sokoldalúbb mint a rendezett listáké.

Subject-Mark reláció

1 tantárgyhoz számos jegy tartozhat

```java
[..]
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<>
  }
  [..]
}
```

Üzleti igény kérdése, de valószínűleg kevésbé hasznos egy tantárgyhoz tartozó jegyeket egyben lekérdezni, így ezen kapcsolat elhagyható. Gyakoribb lehet egy adott diák jegyeinek listázása (pl. féléves összegzésnél). A collection-öket érdemes inicializálni.

Subject.java

CascadeType

A `cascade` értéke egy CascadeType enum value halmaz (`ALL` esetén nem kell felsorolni), mely meghatározza hogy milyen *entity manager* művelet során szükséges a kapcsolatot figyelembe venni, pl.: `cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}`. Alapértelmezésben üres lista.

Mark relációi

Student-Subject N-M kapcsolótábla

```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;
  [..]
}
```

A Date tárolhat időt, dátumot és mindkettőt egyszerre is. Ezt vezérli a @Temporal annotáció.

Cascade

A cascade értelmet szinte kizárólag a **szülő-gyerek asszociáció** során nyer (a szülő entitás állapot változása kihat a gyermek entitásokra). Ellenkező irányban kevésbé hasznos, és sokszor utal *code smell*-re (gyanús hogy nem szándékosan szerepel a kódban, hanem figyelmetlenségből).

CascadeType.PERSIST

Új szülő elem létrehozásakor (beszúrásakor) elegendő csupán a szülőt perisztálni, a benne található gyermek entitások is perzisztálódni fognak.

CascadeType.DELETE

A szülő elem törlése előtt automatikusan a gyermek elemek is törlődni fognak (elegendő a szülőt törölni).

Asszociációk összefoglalása

- `@OneToOne`
  - **Task** (`task_id`) → **TaskDetail** (`taskdetail_task_id`)
  - minden *Task*-nak pontosan egy *TaskDetail*-je van
  - a `cascade = CascadeType.ALL` beállítást egyedül a *Task* entitás esetén alkalmazzuk
  - hasznos lehet az `orphanRemoval = true` opció is (hamis esetén a *Task* id törlésekor/eltűnésekor (pl.: batch update) a *TaskDetail* FK mezője null lesz)
- `@OneToMany` és `@ManyToOne`
  - **Task** (`task_id`) → **SubTask** (`subtask_task_id`)
  - egy *Task*-nak számos *SubTask*-ja lehet
  - *Task*-ban használjuk a `@OneToMany`, míg a *SubTask*-ban a `@ManyToOne` annotációt
- `@ManyToMany`
  - **Bank** (`bank_id`)→ **Account** (`account_bank_id`, `account_client_id`) ← **Client** (`client_id`)
  - Egy *Bank*-nak számos *Client*-je lehet, de egy *Client*-nek is lehet több *Bank*-ban számlája (azonban a példában az *Account*-nak nem lehetnek további adatai (pl. számlaszám...))
  - `CascadeType.ALL` helyett elégséges `PERSIST` + `MERGE` kombinációt használni
  - ebben az esetben az *Account* nem lesz entitás, a `@JoinTable` annotációban fog megjelenni

Hibernate best practice

Ne kössünk be, ne használjunk egzotikus asszociációkat

A gyakorlatban a **many-to-many asszociáció nagyon ritka**. Legtöbb esetben szükségünk van a kapcsolathoz kiegészítő adatokra. Minden ilyen esetben sokkal szerencsésebb két one-to-many asszociációval közre fogni egy osztályt.

Valójában (adatbázis szinten) **minden asszociáció one-to-many vagy many-to-one**. Minden más asszociáció használatát csak óvatosan alkalmazzuk.

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

REST webszolgáltatások web alkalmazása

sch-webservice subproject

```
1  apply plugin: 'war'
2
3  war { archiveName webserviceArchiveName }
4
5  dependencies {
6    providedCompile project(':sch-ejbservice')
7    // providedCompile group: 'javax.servlet', name: 'javax.servlet-api',
         version: servletapiVersion
8    // providedCompile group: 'javax.ws.rs', name: 'javax.ws.rs-api',
         version: jaxrsVersion
9    // providedCompile group: 'javax.ws.rs', name: 'jsr311-api', version:
         '1.1'
10   providedCompile group: 'javax', name: 'javaee-api', version:
         jeeVersion
11 }
```

A *Root* projectben az `application.xml` összeállításakor **school** context root-tal regisztráljuk be a *WEB module*-t.

`build.gradle`

Root project változói:

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

Több lehetőségünk is van a függőség beállítására, a legegyszerűbb a *Java EE 6.0 API*-t használni. Ha a *JSR311 API*-t használjuk, akkor a `SchoolRestApplication` osztályban a `getClasses()` metódust felül kell írni az ősből (`return null;`).

REST Alkalmazás

Application osztály leszármazottja

```
1  package hu.qwaevisz.school.webservice.main;
2
3  import javax.ws.rs.ApplicationPath;
4  import javax.ws.rs.core.Application;
5
6  @ApplicationPath("/api")
7  public class SchoolRestApplication extends Application {
8
9    // @Override
10   // public Set<Class<?>> getClasses() {
11   // return null;
12   // }
13 }
```

Az `@ApplicationPath` annotáció segítségével tudjuk a REST alkalmazás URI-ját beállítani **api**-ra.

A `getClasses()` metódus felülírására csak a *JSR311 API* függőség használata során van szükség.

SchoolRestApplication.java

Student REST szolgáltatás

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

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

  [..]
}
```

Az `@Path` annotáció segítségével tudjuk a REST szolgáltatás URI-ját beállítani **student**-re.

Ugyancsak a `@Path` annotációt tudjuk használni a metódusok szintjén is a REST művelet URI-ját beállítani, a példában pl. **list**-re.

StudentRestService.java

Ha összeolvassuk a beállított URI részeket, akkor megkapjuk a következőt:
http://localhost:8080/school/api/student/list

HTTP Method

Az **HTTP Method**-ok fontos szerepet játszanak a REST szolgáltatások viselkedésében. Gyakori az azonos URI-ra kiküldött kérések HTTP Method alapján való megkülönböztetése a CRUD műveletek megvalósítása érdekében.

- `@POST` → **C**reate
- `@GET` → **R**ead
- `@PUT` → **U**pdate
- `@DELETE` → **D**elete
- `@HEAD`
- `@OPTIONS`

REST műveletek paramétereinek átadása

- `-` (nincs annotációval jelölve)
  - A HTTP Request payload/body elemében kell küldeni az adatot.
- `@QueryParam("ipsum")`
  - `/lorem?ipsum=42&dolor=sit`
- `@PathParam("ipsum")`
  - `/lorem/42/xyz`
  - Ez esetben a `@Path("/lorem/ipsum/xyz")` annotációt kell alkalmazni.
- `@HeaderParam("ipsum")`
  - HTTP Request Header kulcsai között kell lennie egy `ipsum` nevűnek.
  - Speciális esete amikor a **Content-Type**-ot verzéreljük a `@Consumes` annotáción keresztül (a payload-ban megadott adat a megadott MIME type-nak megfelelő)
  - Speciális esete amikor az **Accept**-et vezéreljük a `@Produces` annotáción keresztül (a HTTP Response-ban küldött adat a megadott MIME type szerint elvárt)
- `@CookieParam("ipsum")`
  - HTTP Request Cookie (süti) (browser esetén kényelmes megoldás, de a REST szolgáltatások hívása messze nem korlátozódik böngészőre, így nem ajánlott sütiket alkalmazni RESTful szolgáltatások esetén)
- `@FormParam("ipsum")`
  - Tipikusan *POST*-al küldött `application/x-www-form-urlencoded` MIME type-pal rendelkező kérés esetén hasznos.
  - Erősen kötődik ez esetben a RESTful szolgáltatás egy weboldalhoz. Ha ez kerülendő, ne alkalmazzuk.
- `@MatrixParam("ipsum")`
  - `/lorem;ipsum=42;dolor=sit`
  - Hasonlít a `@QueryParam` esetére, azonban a célja más. Ha egy kulcs-érték nem az egész URI-ra, csak annak pl. egy query param-jára vonatkozik,akkor szokták alkalmazni (paraméter finomhangolás)

Query és Path param lehetőségei

Számos automatizmus segíti fejlesztés közben a paraméterek **type-safe** feldolgozását.

- Értelemszerűen használható `String` típus (hiszen így érkezik)
- Minden primitív típus használható, kivéve a **char** (összekeverhető a String-gel)
- Minden primitív wrapper osztály használható, kivéve a **Character**
- Minden típus, melynek van egyetlen `String` paramétert fogadó constructor-a
- Minden típus, melynek van egy `valueOf(String)` osztályszintű (`static`) metódusa (ennek a szabálynak felel meg minden `enum`)
- `List<T>`, `Set<T>` vagy `SortedSet<T>`, ahol `T` megfelel valamelyik korábban említett feltételnek

Alapértelmezett értékek

A paraméterek átadása nem kötelező (de a hívásoknak egyértelműnek kell lenniük). Ha nem rendelkezik egy paraméter értékkel, akkor primitív esetén *default* lesz (zéró literál), collection esetén üres `List`/`Set` vagy `SortedSet`, minden más esetben pedig `null`. Használható egy `@DefaultValue` annotáció is, melyben felülírható az alapértelmezett érték.

Űrlap feldolgozása

Minta kód

```
1 @POST
2 @Consumes("application/x-www-form-urlencoded")
3 public void post(MultivaluedMap<String, String> formParams) {
4   [..]
5 }
```

Űrlap feldolgozásához egy mezei `Servlet` készítése is legtöbb esetben megfelelő (sokszor célravezetőbb).

URL szabad feldolgozása

@Context annotáció

Teljesen szabad feldolgozása a kapott URL-nek:

```
1 @GET
2 public String get(@Context UriInfo ui) {
3   MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
4   MultivaluedMap<String, String> pathParams = ui.getPathParameters();
5 }
```

Az HTTP Header szabad feldolgozása:

```
1 @GET
2 public String get(@Context HttpHeaders hh) {
3   MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
4   Map<String, Cookie> pathParams = hh.getCookies();
5 }
```

Hasonlóan használható a `HttpServletRequest` és `HttpServletContext` is, illetve mindezeket a REST szolgáltatást implementáló osztályba is be lehet *inject*-álni (nem szükséges az interface-ben szerepeltetni):

```
1 public class Sample {
2
3   @Context
4   private HttpHeaders headers;
5
6   @Context
7   private HttpServletRequest servletRequest;
8   [..]
9 }
```

HTTP Response

Response builder

A REST method visszatérési értéke az HTTP Response payload-ja lesz a beállított MIME type-nak megfelelően. Response visszatérési érték használata során ennél sokkal szabad lehetőségeink is lesznek (bár az interface kevésbé lesz *type-safe*).

```
1 import javax.ws.rs.core.MediaType;
2 import javax.ws.rs.core.Response;
3 import javax.ws.rs.core.Response.Status;
4 [..]
5 Response.ok().build(); // 200 OK
6 Response.noContent().build(); // 204 No Content
7 Response.status(Status.NOT_FOUND).entity([..])
8   .type(MediaType.APPLICATION_JSON).build();
```

Hallgató adatainak lekérdezése

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

Stub vs. Entity

A Stub-ok és Entity-k eleddig nagyrészt megegyeztek, azonban ez nem szükségszerűen van így. A *customer* igények (vagyis a stub-ok) tartalmazhatnak olyan elemeket, melyek pl.:

- Redundánsak, értelemszerűen az entity-k szintjén nem tároljuk (az adatbázis redundancia mentes).
- Ami az entity-k szintjén `A` típusú adat, az a stub-okban `B` típusú (gyakori lehet a típus részletek elfedése és pl. csupán `String` adattípus alkalmazása a stub-ok szintjén (mivel a kliens oldalon úgyis szövegként fog utazni/megjelenni az adat), azonban vegyük figyelembe hogy ezzel *type-safe* viselkedést veszíthetünk.
- A stub-ok szintjén nyelvesített konstansok jelennek meg, vagy egy üzleti logika nyelvi konstansokra alakítja az entity-kben tárolt értékeket (a nyelvi beállítás jöhet a klienstől is, így a visszaküldött adat nyelve kizárólag a klienstől függ, ezzel az adatbázis szintjén nem foglalkozunk).
- A stub-okban olyan mezők is megjelennek, melyek függetlenek a szóban forgó entity-től (mert pl. egy másik rendszer biztosítja, melyet pl. a facade réteg külön kérdez le). Az, hogy az ügyfél által kért adat két helyről érkezik, a kliens oldal előtt nem látható.

Diák adatainak lekérdezése

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

A lekérdezésben a Student entitás mélységi bejárásnak eredménye látható.

A numberOfMarks és a gradeValue számított értékek.


```
{
    "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
}
```

RESTful Endpoint (sch-webservice project)

```
1  @Path("/student")
2  public interface StudentRestService {
3
4    @GET
5    @Path("/{neptun}")
6    @Produces(MediaType.APPLICATION_JSON)
7    StudentStub getStudent(@PathParam("neptun") String neptun)
         throws AdaptorException;
8
9    [..]
10 }
```

StudentRestService.java

Diák adatainak lekérdezése

Top-Down megközelítés

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

- ▷ **sch-webservice**
  - StudentRestService
  - StudentRestServiceBean SLSB
    - Ahhoz hogy az osztályba lehessen EJB-t inject-álni, az EJB context által látottnak kell lennie. Ennek egyik módja hogy SLSB-t készítünk belőle (lehet ezen kívül még CDI-t is használni).
- ▷ **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
    - Számított mezők *accessor* metódusainak elkészítése

```
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();
  }
  [..]
}
```

Generált natív lekérdezés

```
1  SELECT
2    student0_.student_id AS student_1_2_0_,
3    marks1_.mark_id AS mark_id1_0_1_,
4    subject2_.subject_id AS subject_1_3_2_,
5    teacher3_.teacher_id AS teacher_1_4_3_,
6    student0_.student_institute_id AS student_2_2_0_,
7    student0_.student_name AS student_3_2_0_,
8    student0_.student_neptun AS student_4_2_0_,
9    marks1_.mark_date AS mark_dat2_0_1_,
10   marks1_.mark_grade AS mark_gra3_0_1_,
11   marks1_.mark_note AS mark_not4_0_1_,
12   marks1_.mark_student_id AS mark_stu5_0_1_,
13   marks1_.mark_subject_id AS mark_sub6_0_1_,
14   marks1_.mark_student_id AS mark_stu5_2_0__,
15   marks1_.mark_id AS mark_id1_0_0__,
16   subject2_.subject_description AS subject_2_3_2_,
17   subject2_.subject_name AS subject_3_3_2_,
18   subject2_.subject_teacher_id AS subject_4_3_2_,
19   teacher3_.teacher_name AS teacher_2_4_3_,
20   teacher3_.teacher_neptun AS teacher_3_4_3_
21 FROM
22   student student0_
23     LEFT OUTER JOIN mark marks1_ ON
24       student0_.student_id=marks1_.mark_student_id
25     LEFT OUTER JOIN subject subject2_ ON
26       marks1_.mark_subject_id=subject2_.subject_id
27     LEFT OUTER JOIN teacher teacher3_ ON
28       subject2_.subject_teacher_id=teacher3_.teacher_id
29 WHERE
30   student0_.student_neptun=?
```

A *JPQL* lekérdezésben szereplő **FETCH** végett fogja lekérni (és kitölteni) a gyermek entitások adatait (SELECT blokk).

A *JPQL* lekérdezésben szereplő **LEFT JOIN** végett fogja bekötni a gyermek táblákat (FROM blokk). A LEFT-re azért van szükség, mert elképzelhető hogy egy diáknak nincsen jegye, és e nélkül a diák adatai sem jönnének vissza.

Összes hallgató adatainak lekérdezése

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

RESTful Endpoint (sch-webservice project)

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

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

  [..]
}
```

StudentRestService.java

Össszes diák adatainak lekérdezése

Az egész művelet egyszerűen elvégezhető azáltal, hogy a korábban elkészített - egy diák adatait szolgáltató - *named query*-ből kivesszük a neptun kód szűrési feltételt. Most azonban - **bemutott rossz példaként** - nézzük meg mi történik hogyha egy sokkal egyszerűbb *JPQL* lekérdezést készítünk:

```
1 SELECT s
2 FROM Student s
3 ORDER BY s.name
```

Eredmény: `org.hibernate.LazyInitializationException` a `StudentConverterImpl` futása során. Az entitás amit lekértünk és visszaadtunk a *facade rétegnek* **detached** lett, az *entity manager* már nem tud műveleteket végezni rajta, nem tudja felügyelni. Amikor a converter service lekéri a `Student` `getMarks()` metódusát, a container "észleli" hogy itt ha egyszerűen `null`-t adnánk vissza, azzal hamis állapotot idéznénk elő (nem kértük le, így nem tudjuk hogy a diáknak vannak-e jegyei, vagy nincsenek). Mindez kizárólag `LAZY` fetchType mellett fordulhat elő.

Mi lehet a (kerülő) megoldás?

- Írjuk át a fetchType-ot **EAGER**-re: ez a legkényelmesebb megoldás, azonnal működik is a lekérdezés, azzal az "apró" problémával, hogy így az egy lekérdezés helyett immáron **10 lekérdezéssel** állt elő ugyanezen adathalmaz (ráadásul mindez függ attól hogy mennyi különböző tantárgy/tanár van a rendszerben). A megoldás ráadásul borzasztóan pazarol és befolyásol más műveleteket (mivel az entitást módosítottuk).
- Még **attached** állapotban hivatkozzunk a **LAZY** gyermek elemekre (jegyekre), ezáltal kérve az *entity manager*-től az adatok biztosítását. Ezzel a megoldással is **10 lekérdezéssel** leszünk gazdagabbak, de legalább nem befolyásoljuk más műveletek teljesítményét.

```
1  [..]
2  public class StudentServiceImpl implements StudentService {
3    [..]
4    @Override
5    public List<Student> readAll() throws PersistenceServiceException {
6      [..]
7      result = this.entityManager.createNamedQuery(Student.GET_ALL,
           Student.class).getResultList();
8      for (final Student student : result) {
9        student.getMarks().size();
10     }
11     [..]
12   }
13   [..]
14 }
```

Lapozás megvalósítása

Avagy minden esetben a JOIN FETCH az ultimét megoldás?

Egy lista lekérésekor minden esetben felmerülhet az igény arra, hogy ne az összes adatot kérjük le egyszerre, csupán N darabot (pageSize), K offset-tel (page). Erre a natív lekérdezés oldalán pl. az LIMIT és az OFFSET kulcsszavak használhatóak (adatbázis függő lehet). A JPA-ban ezt a TypedQuery/Query példányon tudjuk kiváltani:

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

Figyelem!

Azonban ez nem minden esetben fogja a natív lekérdezésbe elhelyezni a LIMIT kulcsszót, mégis láthatóan működni fog a történet. Hogy lehet mindez? Ha egy entitás gyermek elemet is lekér, akkor az "első K sor" az nem a keresett fő entitás első K sora lesz, hanem pl. az első entitás és gyermekeinek néhány sora (RDBMS szinten ilyenkor a fő entitás sorai ismétlődnek). Ebből következik hogy a JPA - más megoldás nem lévén - lekéri az összes adatot adatbázisból, és Java oldalon választja ki az első K entitást. Ez a megoldás nagy tábla esetén **komoly teljesítmény és erőforrás problémákhoz** vezethet, ráadásul úgy, hogy a fejlesztő optimalizálási céllal vezette be a lapozást.

RESTful Endpoint (sch-webservice project)

```
1  @Path("/student")
2  public interface StudentRestService {
3
4    @GET
5    @Path("/list/{page}")
6    @Produces(MediaType.APPLICATION_JSON)
7    Response getStudents(@DefaultValue("3") @QueryParam("pagesize")
         int pageSize, @PathParam("page") int page) throws
         AdaptorException;
8
9    [..]
10 }
```

StudentRestService.java

Lapozás hatékony megvalósítása

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;
  }
  [..]
}
```

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

```
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
```

Generált lekérdezések

```
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 ( ? , ? , ? , ? )
```

Átlag eredmény statisztika

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

Postman

REST API tesztelésére

- https://www.getpostman.com/
- Verzió: **5.3.2**
- Egyéni használatra ingyenes, csapat munkát támogató funkciók számára létezik egy Pro változat
- Egy `GET` kérést az egyszerűbb esetekben bármely böngészővel egyszerűen tesztelhetünk, ám komplexebb esetekhez apró (X)HTML oldalakat kellene készítenünk, mely nagyon körülményes és nehezen karbantartható.
- Automata tesztekhez valamilyen program nyelven fog készülni script/-forráskód, mely egy profi és programtechnikailag nem jelentős kihívás. Azonban ad-hoc teszteléshez, a fejlesztés támogatásához egy gyorsan üzemkész megoldás célravezetőbb. Ennek egy alternatívája a **Postman**.
- Google account-tal képes szinkronizálni és a projekteket a "felhőben" tárolni

Átlag eredmény statisztika

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

A szolgáltatás egy adott tantárgy (*payload*) vonatkozásában visszaad egy intézményre (*group-by*) és évekre (*group-by*) bontott átlag-eredmény statisztikát (*average*).

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

Van itt valami probléma?!

- Évekre bontva szükséges a jegyeket csoportosítani, de ilyen adat - ilyen formában - nincs meg az adatbázisban. Minden jegynél tároljuk a rögzítés időbélyegét, melyből pl. egy *PostgreSQL* függvénnyel ki tudjuk nyerni a szükséges adatot (`DATE_TRUNC('year', mark_date)` vagy `EXTRACT('year' FROM mark_date)`), azonban JPA szinten itt felmerül sajnos néhány probléma:
  - Különféle dátum függvények léteznek Hibernate-ben és Eclipselink-ben is, ezek egy része használható pl. HQL/EQL-ben, azonban ezeknek egyelőre szabványos formájuk nincsen, vagy azok támogatottsága még nem megfelelő (a most bemutatott példa nem konkrétan a dátum függvényekre vonatkozik, hanem bármilyen olyan speciális függvényre, melyet akár pl. mi hoztunk létre az adatbázisban, és szeretnénk használni ezt JPQL-ben).
  - Akár arra is van megoldás hogy adatbázis függvényeket regisztráljunk JPA szinten, de sokszor ez is implementáció függő (JPA 2.1 támogat olyan megoldást is, mely "felügyelet nélkül" képes meghívni adatbázis oldali függvényeket).
- Egy adott összetettségi ponton egy lekérdezést karbantartani JPQL-ben nehézkes lehet
  - Olyan nem létezik, hogy egy lekérdezést valaki JPQL-ben meg tud írni, de ANSI SQL-ben nem. Mindig (mindig!) először ANSI SQL-ben fogalmazzuk meg a lekérdezéseket, és ezt követően írjuk a JPQL-t (a megfogalmazás történhet fejben is, de e nélkül *nem lehet optimális* JPQL lekérdezéseket készíteni).
  - Az összetett lekérdezések önmagukban is értéket képviselő, algoritmikus részei a forráskódunknak (nyelvben a nyelv), mely megérdemli hogy karbantartható helyen tároljuk → itt jönnek képbe az adatbázis oldali **VIEW**-k.

Nem minden esetben érdemes "erőltetni" a pusztán Java/ORM alapú megoldást. Merjük az egyszerűség jegyében szétdobni a felelősséget az ORM és az RDBMS között.

Natív lekérdezés

```sql
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
```

A tantárgyat az kliens név alapján fogja megadni, a példában a subject tábla bekötése hiányzik a lekérdezés jobb áttekinthetősége végett.

Megoldási terv

Követelmények:

- **group-by** lekérdezést kell készíteni **intézményre** és **évre** nézve
- előzetesen **szűrni** szükséges az adatokat **tantárgyra** nézve

Adatbázis VIEW létrehozása:

- A VIEW-ban előállítjuk azt a mezőt, melyhez adatbázis oldali függvényt szükséges használni (pl.: `DATE_PART`).
- A VIEW-ban minden olyan mezőt szerepeltetni szükséges, melyre igazak az alábbiak:
  - melyre előzetesen szűrést kell végrehajtani (`subject_id`)
  - mely alapján később csoportosítani szükséges az eredményeket (`institute_id` és `mark_year` (számított mező))
  - melyet később fel kell használni az aggregációs függvényben (`mark_grade`)

Figyelem!

Rendkívül ritka az, hogy egy adatbázis VIEW-ban group-by lekérdezés legyen, mivel így későbbiekben semmilyen előzetes szűrést nem lehetne végrehajtani rajta.

Adatbázis VIEW

```
1  CREATE VIEW markdetail AS
2    SELECT
3      ROW_NUMBER() OVER() AS markdetail_id,
4      mark_subject_id AS markdetail_subject_id,
5      student_institute_id AS markdetail_institute_id,
6      mark_grade AS markdetail_grade,
7      DATE_PART('year', mark_date) AS markdetail_year
8    FROM mark
9      INNER JOIN student ON ( mark_student_id = student_id )
10   WHERE ( 1 = 1 );
```

A VIEW-ból entitás lesz ORM szinten, és minden entitásnak kötelező eleme egy elsődleges kulcs. A `ROW_NUMBER()` alkalmas erre (nem fogjuk frissíteni, törölni a *view* egyetlen sorát sem, ráadásul group-by lekérdezés az egyéni sorokat sem fogja visszaadni (így az ID-k értékei memóriában sem jelennek meg majd sehol).

VIEW tesztelése

Ezt a lekérdezést kell ORM szinten megvalósítani

```
1  SELECT
2    markdetail_institute_id,
3    markdetail_year,
4    AVG(markdetail_grade)
5  FROM
6    markdetail
7      INNER JOIN subject ON
8        ( markdetail_subject_id = subject_id )
9  WHERE ( 1 = 1 )
10   AND ( subject_name = 'Sybase PowerBuilder' )
11 GROUP BY
12   markdetail_institute_id,
13   markdetail_year
14 ORDER BY
15   markdetail_institute_id,
16   markdetail_year;
```

VIEW az ORM rétegben

Minden mező pont ugyanúgy van kezelve, mint a többi entitásban. Mi se tegyünk különbséget az ORM rétegben a VIEW és a TABLE között. A `subject` és az `institute` mezők pont ugyanúgy vannak bekötve, mint máshol (ne alkossunk egyedi szabályokat csak azért, mert ezt a VIEW most pl. most egy lekérdezés végett készítettük).

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

  @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 és a generált natív lekérdezés

```
1 SELECT new hu.qwaevisz.school.persistence.result.MarkDetailResult(
2   md.institute,
3   md.year,
4   AVG(md.grade) )
5 FROM MarkDetail md
6 WHERE md.subject.name=:subject
7 GROUP BY md.institute, md.year
8 ORDER BY md.institute, md.year
```

A **JPQL** lekérdezés eredménye egy olyan halmaz, melynek minden eleme egy *intézmény*, egy *évszám* és egy *valós átlag jegy* érték. Ilyen entitás nem létezik az ORM rétegben, ezért ezen adatokat egy erre a célra létrehozott **result**-ba kérdezzük le (`MarkDetailResult`).

```
1  SELECT
2    markdetail0_.markdetail_institute_id AS col_0_0_,
3    markdetail0_.markdetail_year AS col_1_0_,
4    AVG(markdetail0_.markdetail_grade) AS col_2_0_
5  FROM
6    markdetail markdetail0_ CROSS JOIN subject subject1_
7  WHERE
8    markdetail0_.markdetail_subject_id=subject1_.subject_id
9    AND subject1_.subject_name=?
10 GROUP BY
11   markdetail0_.markdetail_institute_id ,
12   markdetail0_.markdetail_year
13 ORDER BY
14   markdetail0_.markdetail_institute_id,
15   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;
  }

  [..]
}
```

Az osztály nem entitás, **egyszerű DTO**. A konstruktor üzletileg fontos, nem szükséges *default* ctor-t készíteni (entitásnál kötelező).

MarkDetailResult.java

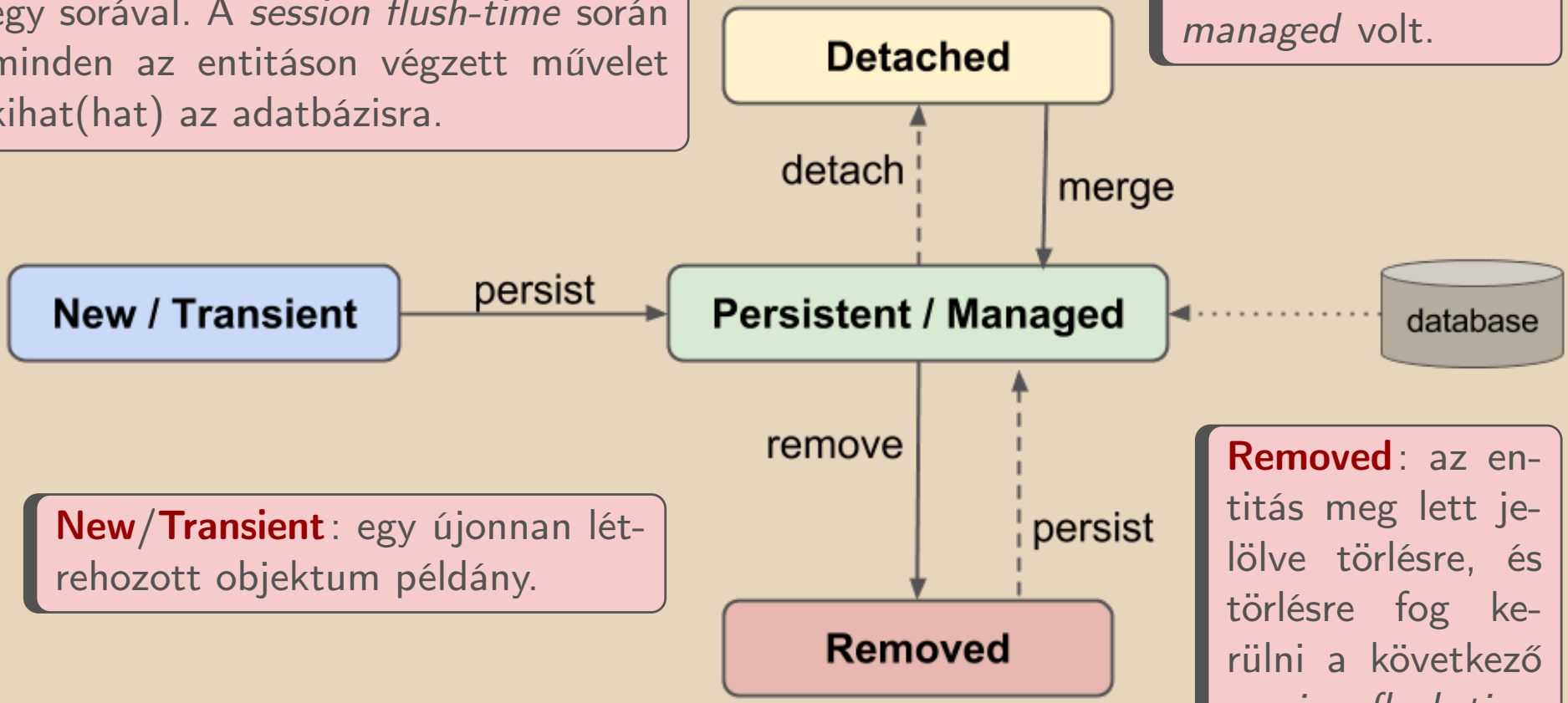
Új érdemjegy rögzítése

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

JPA - Entitás állapotai

Persistent/**Managed**: az aktuális *Persistence Context* számára az adott entitás össze van rendelve ez adatbázis egy sorával. A *session flush-time* során minden az entitáson végzett művelet kihat(hat) az adatbázisra.

Detached: egy entitás, mely korábban *managed* volt.



New/**Transient**: egy újonnan létrehozott objektum példány.

Removed: az entitás meg lett jelölve törlésre, és törlésre fog kerülni a következő *session flush-time* során.

Új érdemjegy rögzítése

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

HTTP Request payload (application/json):

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

A kérésben az érdemjegy egy üzletileg definiált konstansként adott (`WEAK`), mely a perzisztens réteg számára ismeretlen.

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)

```
1  @Path("/mark")
2  public interface MarkRestService {
3
4    @PUT
5    @Path("/add")
6    @Consumes("application/json")
7    @Produces("application/json")
8    MarkStub addMark(MarkInputStub stub) throws AdaptorException;
9
10   [..]
11 }
```

MarkRestService.java

Tranzakciókezelés hiányából eredő problémák

Eleddig lekérdező műveletekkel foglalkoztunk, így "könnyedén" felülemelkedtünk a tranzakciókezelés hiányán, azonban adatmanipuláció során ezt most már nem tudjuk figyelmen kívül hagyni.

- A rögzíteni kívánt jegy tantárgyát egy másik (párhuzamosan futó) tranzakcióban törölhetik a rendszerből, vagy átnevezhetik más nevűre.
- A diákot egy másik (párhuzamosan futó) tranzakcióban törölhetik a rendszerből.

A tranzakciókezelés fontossága üzletileg több megközelítés végett is fontos lehet:

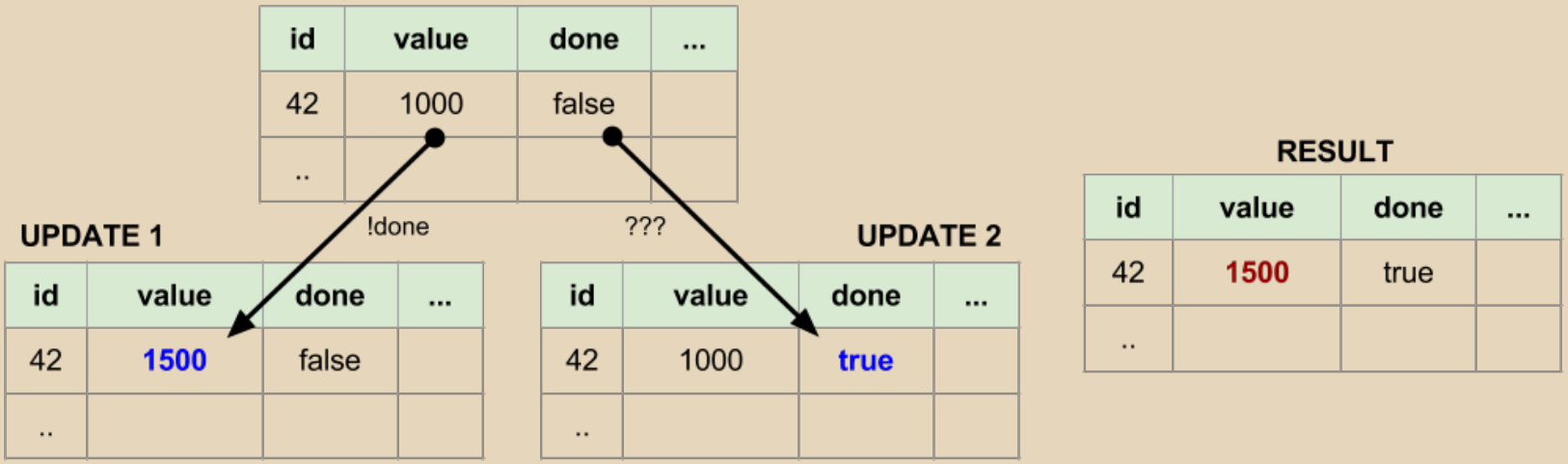
- Elképzelhető hogy a rendszer visszaigazolja hogy a jegy rögzítése sikeres, majd a felhasználó lekérdezi a diák adatait, de már az entitás sem létezik. Mindkét tranzakció sikeresen lefutott, a "program" szempontjából minden tökéletes, mégis úgy érezzük hogy ha a két művelet egyszerre (akár ilyen "helyes" sorrendben) történt, akkor erről a felhasználót tájékoztatni kellene.
- A diák adatait a felhasználó le tudta kérdezni, de mikor a jegyet rögzítené, a rendszer azt üzeni vissza számára hogy a diák nem létezik. A program nem került inkonzisztens állapotba, mégsem lesz elégedett ügyfelünk (az inkonzisztencia elkerülését itt a helyesen normalizált adatbázis fogja biztosítani).

A tranzakciókezelés fontosságát első körben ott tudjuk megfogni, hogy **melyek azok a műveletek** (lekérdezések + adatmanipulációt műveletek), **melyek egy közös tranzakcióban kell hogy végrehajtódjanak**:

- pl. egy új jegy beszúrását megelőzően győződjünk meg arról, hogy ugyanabban a tranzakcióban lekérdezett diák és tantárgy létezik, így még beszúrás előtt tájékoztatni tudjuk a felhasználót arról, hogy miért nem sikerült a rögzítése (ellenkező esetben az `INSERT` elbukik, és a rendszer által visszaadott kivételből *bányásznánk* ki a lehetséges indokot (pl. foreign key sérült, stb.)). **Ha üzletileg értelmezett egy validáció, annak sokszor van helye egy adatmanipuláció előtt.**
- Több új rekordot kell létrehozunk az adatbázis különböző tábláiba (pl. egy *parent* táblába egy sort, és `N` sort a *child* táblába). Ha - akár az utolsó - beszúrás művelete akármilyen okból elbukik, a többi sor sem maradhat az adatbázisban (**rollback** szituáció). Az ORM réteg sokkal jobban összefogja ezt a gyakorlatban (egy ORM akció több adatmanipulációs művelet (is) lehet az RDBMS szintjén), de még ORM szinten is könnyen szembe jöhet egy ilyen szituáció.

Párhuzamosan is sikeres műveletek

Van azonban a tranzakciókezelés kérdéskörében olyan szituáció is, mely során minden helyesen, önálló tranzakcióban hajtódik végre, mégis **üzletileg hibás művelet végrehajtása** következik be, mert a két - külön tranzakcióban bekövetkező - művelet nem akadályozza egymást kölcsönösen!
Ennek leggyakoribb oka, mikor ugyanazon rekordot két különböző személy párhuzamosan *UPDATE*-eli, pl. az egyik *actor* módosítja a rekordban a *fizetendő összeg* mezőt (`value`), a másik *actor* pedig teljesíti/elindítja ezen összeg levonását és beállítja ugyanezen rekord *rendezett flag*-jét (`done`) igazra.



Egyik oldalról meg lehet védeni az `UPDATE 1` műveletet egy validációval (`!done`), de ugyanezt másik irányban nem tudjuk megtenni. Az ilyen szituációkra a **zárolás** (locking) lesz a megoldás.

Lock stratégiák

Pessimistic Locking

Adott adatbázis tábla rekordjának lezárása (locking) annak a tranzakciónak az idejére, amely a lezárást kezdeményezte. Amíg a tranzakció nem jelezte vissza hogy végzett, más művelet nem végezhető el az adott rekordon. A stratégia alkalmazása legtöbb esetben gond nélkül működik, azonban sorosíthatja a beérkező kéréseket, mely könnyen teljesítmény eséshez vezethez (egy nagyon gyakran írt rekord *bottleneck*-je lehet a rendszernek). Figyelni kell arra is, hogy ne alakuljon ki **deadlock** (két tranzakció külön-külön zárol 1-1 rekordot, egyik sem engedi el, miközben mindkét tranzakció sorban áll egy másik rekordért, amit pont a másik tranzakció zárolt).

Optimistic Locking

A korábban bemutatott példában az `UPDATE 2` számára az a probléma, hogy nem tud validációt előzetesen megfogalmazni, mielőtt végrehajtja a `done` flag igazra állítását. Az *actor* számára az volna a fontos, hogy amit korábban lekérdezett, pontosan azon végezze el az *update* utasítást. Ennek egyik módszere lehetne pl. a rekord aktuálisan lekérdezett adatainak *hash*-elése, és közvetlenül az *update* előtt ellenőrizni, hogy még továbbra is egyezik-e a kapott és az újonnan kiszámolt *hash*. Erre léteznek hatékonyabb megoldások is (a *hash* számítás lassú lehet), pl. egy *verzió szám* vagy *időbélyeg* alkalmazása (egy e célra létrehozott plusz oszlop a táblában). Ha ezen validáció elbukik, azt mondjuk hogy a rekord *piszkos* (*dirty*), és elbuktatjuk a tranzakciót. Ott, ahol az adatbázis kapcsolatok pool-ban vannak, ennek a stratégiának a használata javallott.

XA Datasource

eXtended Architecture

Az elosztott tranzakciókezelés standard-ja (***D**istributed **T**ransaction **P**rocessing* (DTP)), mely leírja az interface-t a globális és a lokális *transaction manager* között. Alapvető célja megoldani, hogy különféle erőforrások között ACID[1] tulajdonságú műveletet lehessen végrehajtani (vagyis tranzakciót lehessen commit-álni/rollback-elni akár több adatbázis, vagy egy adatbázis és pl. egy message queue között). Az XA megvalósítása legtöbbször a **kétfázisú tranzakció végrehajtásra** épül.

Kétfázisú tranzakció végrehajtás (two-phase commit, 2PC)

Az **A**tomic **C**ommitment **P**rotocol (ACP) egyik típusa. A tranzakciókat atomic (szét nem választható) egységként kell reprezentálni. Neve onnan ered, hogy egy művelet végrehajtása mindig egy **szavazási** és egy **jóváhagyási** fázisból áll. Az első fázisban minden érintett komponensnek vissza kell jeleznie egy "koordinátor" számára, hogy kész az adott művelet elvégzésére (pl. szabad az erőforrás, elérhető, hálózat rendben van, stb.). Ha minden komponens sikeresen visszajelzett ("igennel" szavazott), akkor a koordinátor felszólítja a második fázisban a komponenseket a tényleges végrehajtásra. A komponensek visszaigazolják végül a műveletet (commit/rollback).

[1] **A**tomicity (atomicitás), **C**onsistency (konzisztencia), **I**solation (izoláció), **D**urability (tartósság)

Tranzakciós jellegzetességek

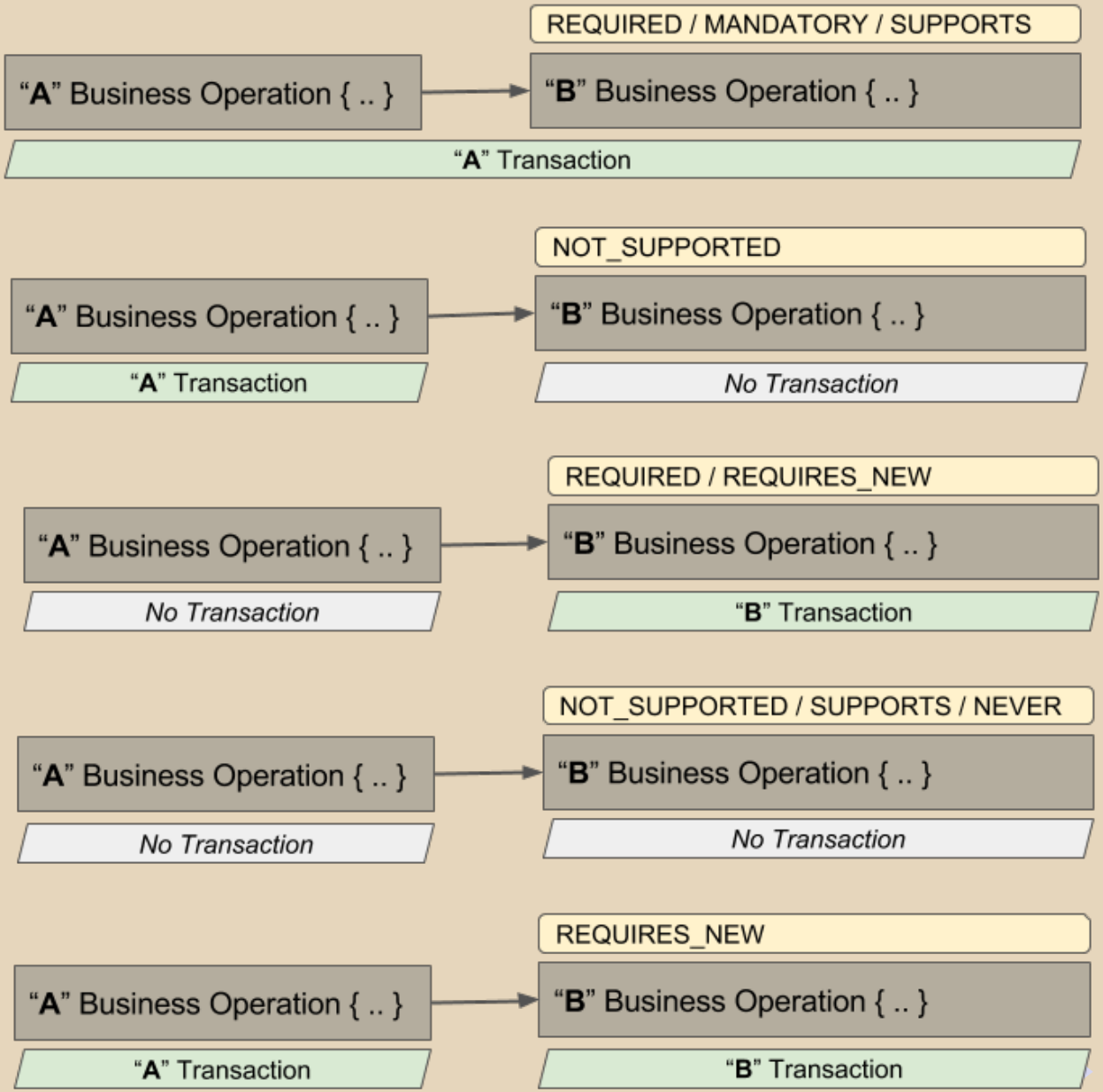
`@TransactionAttribute` annotáció

Ha egy Servlet-ből proxy-n keresztül meghívunk egy EJB service-t, **EJB tranzakció** indulhat. Ezen EJB tranzakciót a szolgáltatás metódusán[2] (vagy az őt tartalmazó osztályon) defininált `@TransactionAttribute` annotáció segítségével lehet konfigurálni.
Kizárólag akkor használható, ha a *container* gondoskodik a tranzakciókezelésről (ez az alapértelmezett, vagy az osztályon definiált a `@TransactionManagement(TransactionManagementType.CONTAINER)` annotáció).
Kliens oldal (hívó üzleti szolgáltatás) hívja az 'távoli' üzleti szolgáltatást. Az annotáció mindig a 'távoli' üzleti szolgáltatáson van:

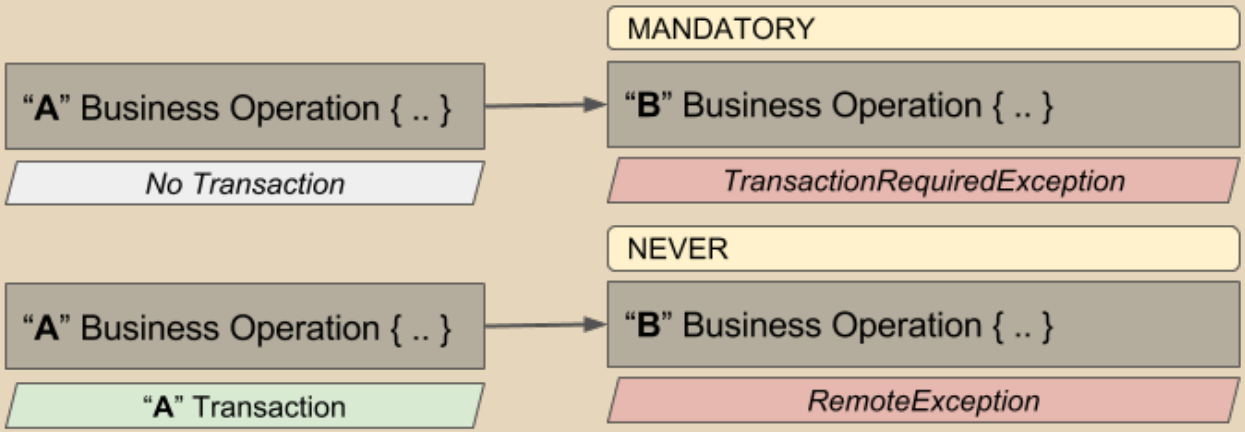
- **MANDATORY**: Kliens oldalon muszáj tranzakcióban futni és az üzleti szolgáltatás ugyanebben a tranzakcióban fog futni.
- **NEVER**: A Kliens oldalnak tilos tranzakcióban futni (ellenkező esetben az üzleti szolgáltatás oldalán hiba lesz).
- **NOT_SUPPORTED**: Az üzleti oldal nem fog tranzakcióban futni.
- **REQUIRED** (alapértelmezett): Ha a kliens oldal tranzakcióban fut, ugyanezen tranzakció fog folytatódni, ellenkező esetben egy új tranzakció jön létre (az üzleti szolgáltatás mindenképpen tranzakcióban fog futni).
- **REQUIRES_NEW**: Az üzleti szolgáltatásnak mindenképpen egy új tranzakcióban kell futnia.
- **SUPPORTS**: Az üzleti szolgáltatás tranzakcióban futása a kliens oldaltól függ (ha a kliens nem fut tranzakcióban, az üzleti oldal sem fog, ha pedig tranzakcióban fut a kliens oldal, akkor az üzleti oldal is). **Különös körültekintés mellett** használjuk csak.

[2] *session bean* vagy *message driven bean* esetén használható

Tranzakciós jellegzetességek - Sikeres esetek



Tranzakciós jellegzetességek - Sikertelen esetek



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());
    }
  }
  [..]
}
```

A `REQUIRES_NEW` megjelölésével garanáltjuk, hogy bárhonnan hívjuk is ezt az üzleti funkciót, az egy önálló tranzakcióban fog végrehajtódni (ha *Servlet*-ből hívjuk akkor e nélkül is ez történne (def. `REQUIRED`)). Minden olyan üzleti funkció, melyet ebből a metódusból hívunk, a `REQUIRED` annotációval fog rendelkezni.

MarkFacadeImpl.java

MarkServiceImpl SLSB (sch-persistent project)

```
1  package hu.qwaevisz.school.persistence.service
2  [..]
3  @Stateless(mappedName = "ejb/markService")
4  @TransactionManagement(TransactionManagementTy
5  public class MarkServiceImpl implements MarkSe
6
7    @PersistenceContext(unitName = "sch-persiste
8    private EntityManager entityManager;
9
10   @Override
11   @TransactionAttribute(TransactionAttributeType.REQUIRED)
12   public Mark create(final Long studentId, final Long subjectId, final Integer grade,
           final String note) throws PersistenceServiceException {
13     try {
14       final Student student = this.entityManager.find(Student.class, studentId);
15       final Subject subject = this.entityManager.find(Subject.class, subjectId);
16       Mark mark = new Mark(student, subject, grade, note);
17       this.entityManager.persist(mark);
18       this.entityManager.flush();
19       return mark;
20     } catch (final Exception e) {
21       throw new PersistenceServiceException("..." + e.getLocalizedMessage(), e);
22     }
23   }
24   [..]
25 }
```

Csak **attached** (managed) entitást lehet menteni (`persist` vagy `merge`). Az ID ismeretében (melyekkel már rendelkezünk) az *entity manager* `find()` műveletével könnyedén készíthetünk *csatolt* entitásokat (ez jelen esetben nem fog új lekérdezést sem generálni tranzakción belül).

Figyelni kell arra, hogy **soha ne csatoljuk ugyanazt az entitást kétszer**. Ez esetben "*Multiple representations of the same entity are being merged.*" hibát fogunk kapni (ha nem tudjuk elkerülni, a `CascadeType.MERGE`/`CascadeType.PERSIST` lehetőséget kell elvennünk valahol az ORM rétegben).

Generált natív lekérdezések

```
SELECT
  subject0_.subject_id AS subject_1_3_,
  [..]
  subject0_.subject_teacher_id AS
       subject_4_3_
FROM subject subject0_
WHERE s
SELECT
  teacher0_.teacher_id AS teacher_1_4_0_,
  teacher0_.teacher_name AS
       teacher_2_4_0_,
  teacher0_.teacher_neptun AS
       teacher_3_4_0_
FROM teacher teacher0_
WHERE teacher0_.teacher_id=?
```

Subject validációja: **2 SELECT** (Subject.teacher EAGER).

```
SELECT
  student0_.student_id AS
       student_1_2_0_,
  marks1_.ma
  [..]
  teacher3_.
       teacher_3_4_3_
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=?
```

Student validációja: **1 SELECT** (optimalizált).

```
SELECT
  NEXTVAL ('mark_mark_id_seq')

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

Mark beszúrása: **1 SELECT + 1 INSERT**.

Hallgató törlése

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

Hallgató törlése

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)

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

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

  [..]
}
```

StudentRestService.java

HTTP státusz kódok

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

Hibakezelés RESTful interface-en

- Az *HTTP Response status code* field-je alapján különböző "payload"-ot küldhetünk vissza, hiszen ezt bármilyen fogadó kliens alkalmazás könnyedén feldolgozza és szétválasztja. Hiba esetén egy `ErrorStub` példány `JSON` formátumban tartalmazhatja a hiba üzleti kódját, esetleg további **publikus információkat**. Szabványos megoldás (mint pl. a SOAP Fault) nincsen.
- Egy objektum-orientált rendszerben egy üzleti metódus hibaesetei (és ezáltal "különböző" visszatérési értékei) a kivételkezelés segítségével válnak elkülöníthetővé és kezelhetővé. Ha üzleti hiba történik, általában *checked* kivételt dobunk (`AdaptorException` példány), mely tartalmazhat a publikus információk mellett a hiba felderítést megkönnyítő **védett adatokat is**. A kivételhez a `JAX-RS`-ben egy `ExceptionMapper<T>` hozható létre, mely transzformálja az üzleti hibát a szükséges *HTTP Response*-ra (`AdaptorExceptionMapper`).

A fentiekből következik, hogy az `AdaptorException` lesz az `ErrorStub` **factory**-ja! Programtechnikailag azonban a hiba keletkezésekor megadni minden szükséges adatot az `ErrorStub` számára nem volna szerencsés, hiszen a publikus hibaüzenet sokszor ismétlődik (tömörít), ezáltal a kódban komoly redundancia keletkezne, melyet nehéz volna karbantartani (pl. megváltozik az üzleti hibakód az egyik általános hibaeset során). A fenti elkerülése végett definiáljunk egy `ApplicationError` *enum*-ot, mely egységbe zárja az `ErrorStub` redundáns, ismétlődő részeit. Ezáltal a hiba keletkezésekor csak ezen enum egy példányát, illetve a hibaüzenet változó elemeit szükséges megadnunk.

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);
  }
}
```

Az ApplicationError enum példány az ErrorStub factory-ja, illetve képes visszaadni (és tárolni) az HTTP Status értékét is, mely szintén korrelál a hiba típusával.

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);
  }
}
```

Az AdaptorException példány az ErrorStub factory-ja (a feladatot delegálja az ApplicationError enum példány számára).

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();
  }

}
```

A `@Provider` osztályok konfigurációs lehetőséget biztosítanak a JAX-RS számára. Számos ilyet tartalmaz a JAX-RS implementáció is (pl. object-XML/JSON kétirányú konverzió).

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());
    }
  }
}
```

Mind a három persistent művelet tranzakciós attribútuma REQUIRED, így a lekérdezések és maga a törlés egy tranzakcióban történik.

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 lekérdezések

Létezik-e a hallgató?

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

Ha létezik, vannak-e jegyei?

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

Ha nincsenek, akkor a törlés végrehajtása

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

Cross-Origin Resource Sharing (CORS)

A CORS egy technika arra hogy a böngészők (*user agent*) engedélyt kérjenek ahhoz hogy HTTP kéréseket küldhessenek (és fogadhassanak) egy *más* domainnal rendelkező szolgáltatástól (a *más* itt az eredetileg meghívott domaintől eltérőt jelenti).
A böngésző ilyen esetben egy `OPTION` HTTP kérést küld a szervernek (az eredeti kérés HEADER (és url) adataival), ezzel kérve az adott szolgáltatás meghívásának engedélyét. **A szerver oldali komponens dolga ezen `OPTION` kérés feldolgozása, és eldöntése hogy pl. adott IP címmel rendelkező kliens meghívhatja-e a szolgáltatást**. Nagyon gyakori, hogy a CORS filter szerver oldalon úgy működik, hogy minden helyről érkező HTTP kérést engedélyez. Ezt fogjuk mi is most alkalmazni. Ha a szerver oldal elutasítja a kérést, a *user agent* nem fogja az eredeti kérést elküldeni.

Léteznek 3rd party megoldások CORS filterekre, de mi most ezek nélkül eszközlünk egy megoldást.

CORS - OPTION kérések feldolgozása

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

Alapértelmezett debug port: **8787**

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

server.log

Bármely JVM-et lehet remote debug-olni, csupán az indító java parancsnak kell az alábbi argumentumokat átadni (az -Xdebug a régebbi JVM beállítása, de az újabbak is felismerik):

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

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

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

startWebLogic.cmd

Debug port: **4000**

- -XDebug → debug engedélyezése
- -Xnoagent → az alapértelmezett sun.tools.debug debug *agent* kikapcsolása
- -Xrunjdwp → a JDWP[3] JPDA[4] referencia implementációjának betöltése

[3] Java™ Debug Wire Protocol
[4] Java™ Platform Debugger Architecture

Run | Debug Configurations |

- ▷ Remote Java Application
  - Helyi menü: New
  - Project: Browse.. (egyébként ez lényegtelen)
  - Connection Type: Standard (Socket Attach)
    - Host: **localhost**
    - Port: **8787**
  - Apply és Debug
- ▷ Debug Perspective-re váltás
  - Debug view-ban látni kell a thread-eket
  - Ugyanitt: Helyi menü: Edit source lookup
    - Add Java Project(s)

Egység tesztek készítése

Egység tesztelés

Bedők Dávid: **Programozási feladatok megoldási módszertana** (Óbudai Egyetem, 2015)
5.2 fejezet: Egység tesztelés

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

Az egység teszt készítés célját és szerepét ez a labor nem érinti. A **TestNG** 3rd party library használata már korábban - nagyon felületesen - érintve volt. A hangsúly most az **EJB service**-ek egység tesztelésén van. Miként és hogyan tesztelünk olyan osztályokat, ahol az osztályok által használt erőforrásokat (pl.: más EJB/CDI bean-eket (azok proxy-jait), resource-okat) egy container vagy framework inject-álja az osztályba futási időben. E kapcsán ismerjük meg a **Mockito** 3rd party library-t (mock/fake osztályok készítését könnyíti meg).

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

Miért mockoljunk?

Elsősorban azért, mert ha valós osztályként használnánk azokat, akkor ha a felhasznált osztályban hiba van, akkor nem csak annak az egység tesztje bukna el, hanem azok az osztályoknak az egység tesztjei is, melyek őt felhasználják.
Nem minden függőséget szükséges mockolni, kivételek mindig előfordulhatnak. Ezt megfelelő egység teszt írási tapasztalat után a szakember érezni fogja.

Átlag jegy statisztika egység tesztelése

sch-ejbservice project

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

Mi a metódus felelőssége ebben a retégben (whitebox tesztelés[5])?

- ▷ Egy bemeneti tantárgy neve alapján előállítani egy kimeneti `MarkDetailStub` listát.
- ▷ A tárgy neve alapján a persistence rétegtől elkérni a statisztikát tartalmazó adatokat (`MarkDetailResult` lista)
- ▷ A perzisztens rétegből visszakapott lista átalakítását kell kérni egy e célra szolgáló konverziós szolgáltatástól, hogy előálljon a hívó számára értelmezhető `MarkDetailStub` lista.
- ▷ Ha a perzisztens rétegben hiba keletkezik, akkor nem várt hibát szükséges a hívónak jelezni (`ApplicationError.UNEXPECTED`).
- ▷ Ha az adott tárgy nem létezik, üres listát kapunk vissza.

A felsoroltakon kívül nincs más felelőssége az üzleti metódusnak (pl. nem érdekli hogy miként zajlik a konverzió, hogy milyen *named query* hajtódik végre a persistence rétegben és hogy van-e adatbázis oldali VIEW e mögött

MarkFacadeImplTest (sch-ejbservice project)

Teszt előkészítése

```
1  package hu.qwaevisz.school.ejbservice.facade;
2  [..]
3  public class MarkFacadeImplTest {
4
5    @InjectMocks
6    private MarkFacadeImpl facade;
7
8    @Mock
9    private MarkService markService;
10
11   @Mock
12   private MarkConverter markConverter;
13
14   @BeforeMethod
15   public void setup() {
16     MockitoAnnotations.initMocks(this);
17   }
18
19   [..]
20 }
```

Az `@InjectMocks` annotációval az az egyetlen osztály rendelkezik, melyet az adott egység tesztben tesztelünk. Ide kell a mockokat a keretrendszernek "inject"-álnia. A `MarkFacadeImpl` példányt **ne példányosítsuk**!

A `@Mock` annotációval azok az osztályok szerepelnek, melyeknek egy mock-ot kell gyártani, és melyek be lesznek inject-álva a tesztelendő osztályba. A mockokat használat előtt többnyire elő kell készíteni (a fake-ek ezzel szemben "készre" készülnek).

A `MockitoAnnotataions.initMocks(this)` nagyon fontos, hogy minden teszt metódus előtt lefusson. Ez végzi el az inject-álást. A tesztek ős osztályba áthelyezhető a kódsor.

MarkFacadeImplTest.java

MarkFacadeImplTest (sch-ejbservice project)

Nincsenek statisztikai adatok

```
1  public class MarkFacadeImplTest {
2    [..]
3    private static final String SUBJECT_NAME = "LoremIpsumSubject";
4
5    @Test
6    public void returnAnEmptyListWhenTheSubjectIsNotExistsOrHasntGotAnyGrades() throws
         AdaptorException, PersistenceServiceException {
7      final List<MarkDetailResult> results = new ArrayList<>();
8      Mockito.when(this.markService.read(SUBJECT_NAME)).thenReturn(results);
9      final List<MarkDetailStub> stubs = new ArrayList<>();
10     Mockito.when(this.markConverter.to(results)).thenReturn(stubs);
11
12     final List<MarkDetailStub> markDetailStubs =
           this.facade.getMarkDetails(SUBJECT_NAME);
13
14     Assert.assertEquals(markDetailStubs.size(), 0);
15   }
16   [..]
17 }
```

MarkFacadeImplTest.java

Annak követelménynek kell lennie, hogy `markService.read()` üres listát ad vissza abban az esetben, ha az adott tárgyhoz nem tartoznak jegyek, vagy ha nem is létezik. A teszt azt ellenőrzi, hogy ez esetben nem fut hibára a `getMarkDetails()` metódus.

MarkFacadeImplTest (sch-ejbservice project)

Sikeres eset

```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)

Sikertelen eset

```
1  public class MarkFacadeImplTest {
2    [..]
3
4    @Test(expectedExceptions = AdaptorException.class)
5    public void
           throwUnexpectedApplicarionErrorIfSomethingErrorOccoursInThePersistenceLayer()
           throws PersistenceServiceException, AdaptorException {
6      Mockito.when(this.markService.read(SUBJECT_NAME)).thenThrow(PersistenceServiceException
7      this.facade.getMarkDetails(SUBJECT_NAME);
8      Assert.fail();
9    }
10
11   [..]
12 }
```

MarkFacadeImplTest.java

A dobott AdaptorException egyik mezője (ApplicationError enum) tartalmaz javax.ws.rs.core.Response.Status példányokat. E miatt a teszt *classpath*-ra el kell helyezni (pl.) a org.jboss.spec:jboss-javaee-6.0 artifact-ot (e miatt Gradle esetén használjuk az JavaEE API esetén a compileOnly *dependency configuration*-t a compile helyett).

Tipikus forgatókönyv

```
Subject subject = Mockito.mock(Subject.class);
```
Létrehoz egy `Subject` mock-ot (a `@Mock` annotáció is ilyet hoz létre, azonban utóbbit inject-álja is a tesztelendő osztályba, ha erre kérjük).

```
Mockito.when(this.markService.read(SUBJECT_NAME))
    .thenReturn(results);
```
Felkészít egy mock-ot. Jelen esetben ha a `read()` metódusát egy adott `String` paraméterrel meghívjuk, vissza ad egy `results`-et (ami jelen esetben egy listányi mock, de lehet valós osztálypéldány/literál is).

```
Mockito.verify(this.markService).read(SUBJECT_NAME);
```
Ellenőrzi a tesztelendő metódus meghívása után a `read()` metódus meghívását a service mock-ján a megadott `String` példány paraméterrel. Ha nem hívódik meg (pontosan egyszer), akkor a teszt elbukik, mivel a hívás elvárt!

Mockito további lehetőségei

- Lehetőség van `when()` során kivétel dobására (utóbbit a `void` visszatérési értékű metódusok is megtehetik).
- Van lehetőség `Matcher`-ek segítségével nem pontos értéket átadni paramétereknek, hanem pl. csak az a fontos hogy `String` osztály példánya legyen. Tetszőlegesen kombinálható mindez.
- Tudunk belső argumentumokat "elkapni" (mivel hívták meg a *mock* metódusát), majd az egység tesztben erre pl. egy `Assert`-et írni.
- Megadható pontosan hányszor hívtak meg egy metódust `verify()` során.
- Megadható `when()` során ha ugyanazt a metódus többször hívják, sorban miket adjon vissza eredményül.
- stb.

Do not overengineering

Természetesen a Mockito osztálykönyvtár/library számos egyéb lehetőséget tartalmaz, azonban nem szabad megfeledkezni arról sem, ha túlságosan "mélyen" tesztelünk a vizsgált osztályt, akkor az nagyon érzékeny lesz az apróbb módosításokra is (nehezebben lesz refaktorálható). E miatt pl. a `verify()` használatát ahol lehet mellőzzük (a bemutatott példában pl. teljesen szükségtelen).

Diák jegyeinek lekérdezése szűrési feltételekkel

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

Szűrt eredmények listája

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>
```

A szolgáltatás segítségével a tantárgy nevének egy részletével (`subject`) illetve az érdemjegyek alsó- (`minimumgrade`) és felső (`maximumgrade`) határának megadásával lehetséges a megadott diák (`neptun`) szerzett jegyeit listázni.

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>
```

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

A org.jboss.resteasy.annotations.providers.jaxb.Wrapped annotáció segítségével a List<T> befoglaló element neve definiálható (ez XML esetén értelmezhető, JSON esetén nem). Az annotáció használata miatt fordítási függőségek közé szükséges felvenni a org.jboss.resteasy:resteasy-jaxb-provider artifact-ot.

JPQL és a generált natív lekérdezés

```
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 alkalmazás

```
1  private static final String REQUEST_PAYLOAD = "" //
2      + "<markcriteria>" //
3      + "  <subject>Programming</subject>" //
4      + "  <minimumgrade>1</minimumgrade>" //
5      + "  <maximumgrade>3</maximumgrade>" //
6      + "</markcriteria>";
7  public static void main(String[] args) throws IOException {
8    URL url = new URL("http://localhost:8080/school/api/student/marks/WI53085");
9    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
10   connection.setRequestMethod("POST");
11   connection.setRequestProperty("Content-Type", "application/xml");
12   connection.setUseCaches(false);
13   connection.setDoOutput(true);
14   DataOutputStream outputStream = new DataOutputStream(connection.getOutputStream());
15   outputStream.writeBytes(REQUEST_PAYLOAD);
16   outputStream.close();
17
18   InputStream inputStream = connection.getInputStream();
19   BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
20   StringBuilder response = new StringBuilder();
21   String line;
22   while ((line = reader.readLine()) != null) {
23     response.append(line);
24   }
25   reader.close();
26
27   System.out.println(response);
28 }
```

A RESTful service-ek meghívása teljesen nyelvfüggetlen, HTTP Request-ek gyártása kell csupán hozzá, "szinte" bármilyen programozási nyelvből lehetséges. A bemutatott Java példa elkészíti a POST kérést és feldolgozza a választ. Az XML válasz természetesen szöveges formában fog rendelkezésre állni.

A `java.net` csomag használatával az a legnagyobb probléma, hogy nem ad *type-safe* megoldást és számos *boilerplate* kódot eredményez. Léteznek természetesen 3rd party library-k, melyek ezen próbálnak javítani, de szerencsére a JAX-RS népszerű implementációi is adnak kliens oldali megoldást, vagyis ezen könyvtárak kliens oldali kód esetén Java SE alkalmazásban is használhatóak (itt most a JBoss **RESTeasy** és az Oracle **Jersey**-re gondolva első sorban).

Párban használjuk?

Nincs jelentősége annak hogy kliens oldalon milyen 3rd party könyvtárat használunk. Attól hogy a server oldalon RESTeasy van, a kliens oldalon lehet Jersey implementáció. Annak sincs jelentősége hogy milyen *stub*-okat használunk, csupán az a fontos hogy azok a megfelelő MIME type szerint legyenek szerializálva (azt az XML/JSON tartalmat állítsák elő amire szükség van).

JAXB

A **JAXB Provider** (**J**ava **A**rchitecture for **X**ML **B**inding) az XML-ek szerializálásához és deszerializálásához szükséges.

```
1 jar { archiveName 'sch-restclient.jar' }
2
3 dependencies {
4   compile group: 'org.jboss.spec', name: 'jboss-javaee-6.0',
        version: jbossjee6Version
5   compile group: 'org.jboss.resteasy', name:'resteasy-jaxrs',
        version: resteasyVersion
6   compile group: 'org.jboss.resteasy',
        name:'resteasy-jaxb-provider', version: resteasyVersion
7   compile group: 'commons-logging', name: 'commons-logging',
        version: commonsloggingVersion
8 }
```

build.gradle

```
1 ext {
2   jbossjee6Version = '3.0.3.Final'
3   resteasyVersion = '2.3.7.Final'
4   commonsloggingVersion = '1.2'
5 }
```

RESTful Remote Endpoint (sch-restclient project)

```
1  package hu.qwaevisz.school.restclient;
2  [..]
3  @Path("/student")
4  public interface StudentRemoteRestService {
5
6    @POST
7    @Consumes("application/xml")
8    @Produces("application/xml")
9    @Path("/marks/{student}")
10   @Wrapped(element = "marks")
11   ClientResponse<List<MarkStub>>
         getFilteredMarks(@PathParam("student") String neptun,
         MarkConditions conditions);
12 }
```

StudentRemoteRestService.java

A StudentRemoteRestService néhány ponton szándékosan (prezentációs céllal) eltér a szerver oldali StudentRestService-től (pl. az elérési útban student szerepel, és a MarkCriteria osztály neve is más kliens oldalon). A ClientResponse<T> használata praktikus, mert így a *HTTP Response header*-jét/*response code*-ját is elérjük, nem csupán a visszakapott *entity*-t.

Példakód (sch-ejbservice project)

```
1  public List<MarkStub> process(String studentNeptun,
       MarkConditions conditions) {
2    URI serviceUri =
         UriBuilder.fromUri("http://localhost:8080/school/api").build();
3    ClientRequestFactory crf = new ClientRequestFactory(serviceUri);
4
5    StudentRemoteRestService api =
         crf.createProxy(StudentRemoteRestService.class);
6    ClientResponse<List<MarkStub>> response =
         api.getFilteredMarks(studentNeptun, conditions);
7
8    LOGGER.info("Response status: " + response.getStatus());
9    MultivaluedMap<String, Object> header = response.getMetadata();
10   for (final String key : header.keySet()) {
11     LOGGER.info("HEADER - key: " + key + ", value: " +
           header.get(key));
12   }
13   List<MarkStub> marks = response.getEntity();
14   return marks;
15 }
```

SchoolRestClient.java