MONASH University

Information Technology

FIT5183: Mobile and Distributed Computing Systems (MDCS)

Lecture 4B – Java Persistence Query Language (JPQL) and Named Queries

Outline

- ❑ Entity Modelling and Relationships
- ❑ Private and Foreign Keys (PK, FK)
- ❑ Java Persistence Query Language (JPQL)
- ❑ Summary of JAX-RS Annotations
- ❑ Static and Dynamic [Named or Typed] Queries
- ❑ Implicit and Explicit Join

Data Entity Models and Relationships

Figure : Attributes

Entity Relationship (ER) modelling

ER modelling is a design tool

- **ER diagrams** are a graphical representation of the database system providing a high-level conceptual data model
- supports the user's perception of the data
- is DBMS and hardware independent and has many variants
- is composed of **entities**, **attributes**, and **relationships**

Entities

- An entity is any object in the system that we want to model and store information about Individual objects are called entities
- Groups of the same type of objects are called entity types or entity sets
- Entities are represented by ***rectangles*** (either with round or square corners)
- There are two types of entities; weak and strong entity types.

Source http://db.grussell.org/section004.html

Figure : Attributes

ER modelling - continued

Attribute

- All the data relating to an entity is held in its attributes
- Attributes are the properties of an entity.
- Each entity within an entity type may have any number of attributes (keep diagram simple)
- Attributes can be simple or composite
- They appear inside ***ovals*** and are attached to their entity

Relationships

- A relationship type is a meaningful association between entity types
- A relationship is an association of entities where the association includes one entity from each participating entity type.
- Relationship types are represented on the ER diagram by a ***series of lines*** (in the original Chen notation, the relationship is placed inside a diamond.)

Source http://db.grussell.org/section004.html

ER modelling - continued

Keys

- Keys are data items that allow us to **uniquely identify individual occurrences of entity types**.
- A candidate key is an attribute or set of attributes that uniquely identifies individual occurrences of an entity type.
- An entity type may have one or more possible candidate keys, the one which is selected is known as the ***primary key*** **(PK).**
- A composite key is a candidate key that consists of two or more attributes.
- The name of each primary key attribute is ***underlined.***
- A ***foreign key*** **(FK)** is an attribute (or composite) that is the primary key to another relation.
- Roughly, each foreign key represents a relationship between two entity types.
- They allow the relations to be linked together.
- A relation can have several foreign keys.
- Foreign keys are usually shown in italics or with a ***wiggly underline***.

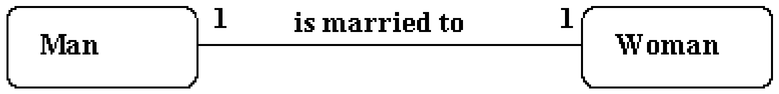
Source http://db.grussell.org/section004.html

.

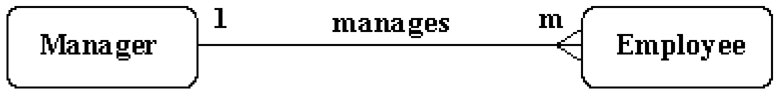
Cardinality in Entity Relations

“Relationships are rarely one-to-one. For example, a manager usually manages more than one employee. This is described by the *cardinality* of the relationship, for which there are four possible categories.”

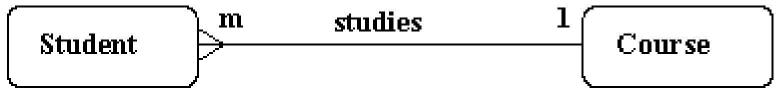
- ❑ One to one (1:1) relationship



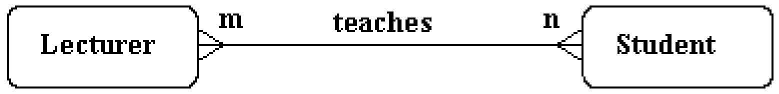
- ❑ One to many (1:m) relationship



- ❑ Many to one (m:1) relationship



- ❑ Many to many (m:n) relationship

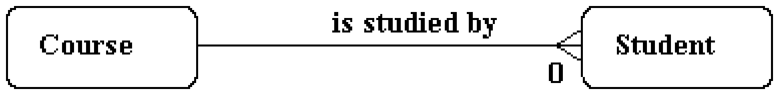


On an ER diagram, if the end of a relationship is straight, it represents 1, while a "crow's foot" end represents many.

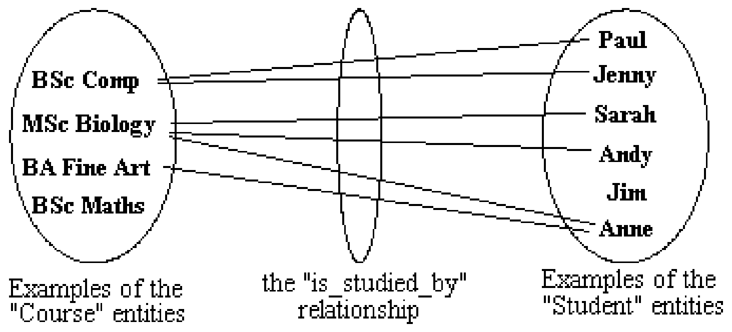
Source http://db.grussell.org/section004.html

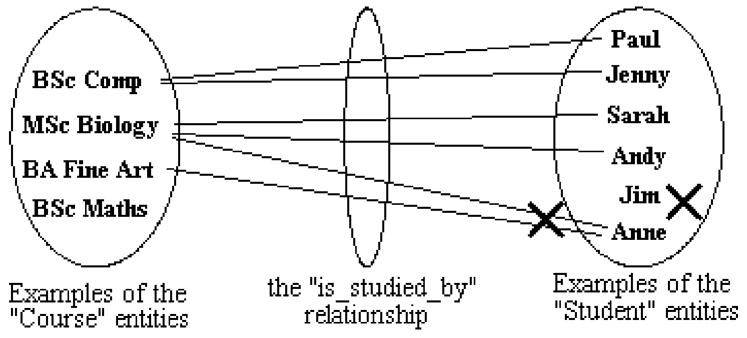
ER Optionality and Entity Sets

A relationship can be optional or mandatory. If the relationship is mandatory an entity at one end of the relationship must be related to an entity at the other end.



- The optionality can be different at each end of the relationship
- For example, a student must be on a course. So the relationship `student_studies_course' is mandatory.
- But a course can exist before any students have enrolled. Thus the relationship `course_is_studied_by student' is optional.
- To show optionality, put a circle or `0' at the `optional end' of the relationship.
- Use Entity Set diagrams to show cardinality and optionality under all possible relationship scenarios


BSc Comp
MSc Biology
BA Fine Art
BSc Maths
Paul
Jenny
Sarah
Andy
Jim
Anne
Examples of the "Course" entities
the "is_studied_by" relationship
Examples of the "Student" entities


BSc Comp
MSc Biology
BA Fine Art
BSc Maths
Paul
Jenny
Sarah
Andy
Jim
Anne
Examples of the "Course" entities
the "is_studied_by" relationship
Examples of the "Student" entities

Source http://db.grussell.org/section004.html

Constructing an ER model

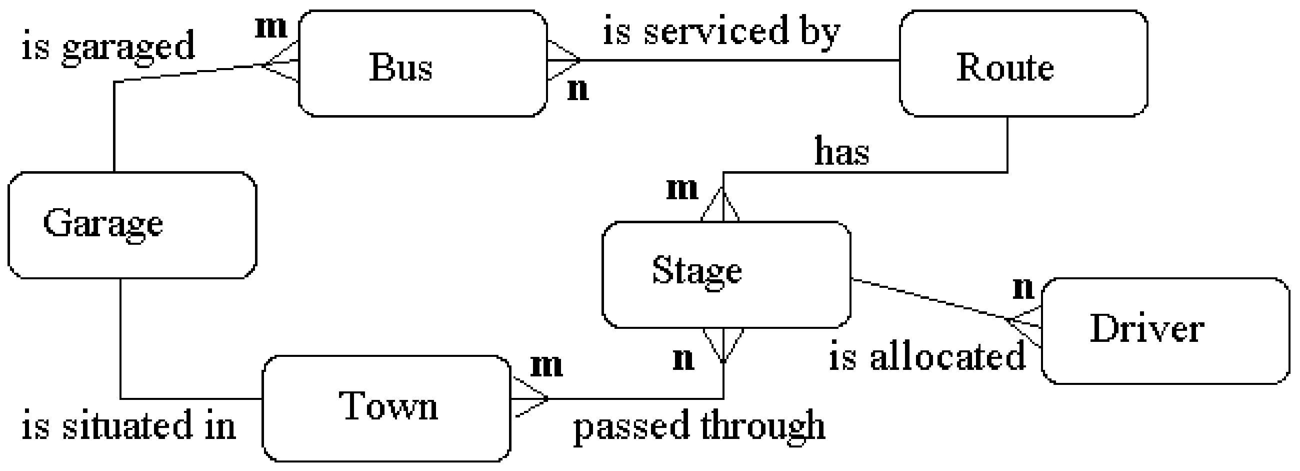
Before beginning to draw the ER model, read the requirements specification carefully. Document any assumptions you need to make.

1. **Identify entities** - list all potential entity types. These are the objects of interest in the system. It is better to put too many entities in at first and discard them later if necessary.
2. **Remove duplicate entities** - Ensure that they really separate entity types and are not just two names for the same thing.
1. **List the attributes** of each entity (all properties to describe the entity which are relevant to the application). Ensure that the entity types are really needed and are not attributes of another entity type.
3. **Mark the primary keys**. Which attributes uniquely identify instances of that entity type? This may not be possible for some weak entities.
4. **Define the relationships**. Examine each entity type to see its relationship to the others.
5. **Describe the cardinality and optionality** of the relationships. Examine the constraints between participating entities.
6. **Remove redundant relationships**. Examine the ER model for redundant relationships.

ER modelling is an iterative process, so draw several versions, refining each one until you are happy with it. Note that there is no one right answer to the problem, but some solutions are better than others!

Example ER Diagram

System/Problem Description: *A Country Bus Company owns a number of busses. Each bus is allocated to a particular route, although some routes may have several busses. Each route passes through a number of towns. One or more drivers are allocated to each stage of a route, which corresponds to a journey through some or all of the towns on a route. Some of the towns have a garage where busses are kept and each of the busses are identified by the registration number and can carry different numbers of passengers, since the vehicles vary in size and can be single or double-decked. Each route is identified by a route number and information is available on the average number of passengers carried per day for each route. Drivers have an employee number, name, address, and sometimes a telephone number*.

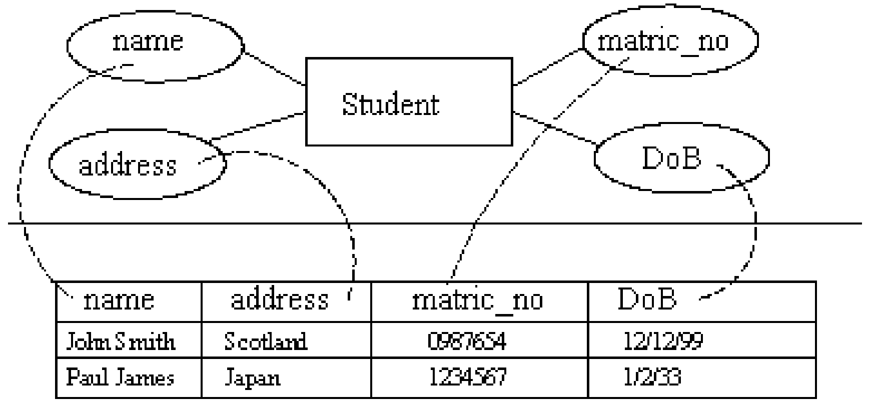


Source http://db.grussell.org/section004.html

Mapping ER Models into Relations

A relation is a table that holds the data we are interested in. It is 2-dimensional and has rows and columns. Each entity type in the ER model is mapped into a relation.

- The attributes become the columns.
- The individual entities become the rows.



name	address	matric_no	DoB
John Smith	Scotland	0987654	12/12/99
Paul James	Japan	1234567	1/2/33

Source http://db.grussell.org/section004.html

- Relations can be represented textually as:

tablename(<u>primary key</u>, attribute 1, attribute 2, ... , *foreign key*)

Cardinality corresponds to the number of rows and -arity is the number of columns in the table.

Java Persistence Query Language (JPQL) and Java Persistance API (JPA)

JPQL (Java Persistence Query Language)

- ❑ JPQL is an object-oriented query language
- ❑ JPQL is part of the Java Persistence API (JPA) specification
  - JPA provides an object/relational mapping facility for managing relational data
- ❑ JPQL syntax is similar to SQL
- ❑ JPQL makes queries on **persistent entity objects** (i.e. entities) and their persistent state
- ❑ JPQL queries operate on **entities** instead of directly on the database tables (e.g. dbs tables in jdbc:derby)

SQL Statements and Clauses

- **SELECT** Statement: **SELECT** column1, column2....columnN FROM table_name;
- **DISTINCT** Clause**:** SELECT **DISTINCT** column1, column2....columnN FROM table_name;
- **WHERE** Clause: SELECT column1, column2....columnN FROM table_name **WHERE** CONDITION;
- **AND/OR** Clause**:** SELECT column1, column2....columnN FROM table_name WHERE CONDITION-1 {**AND|OR**} CONDITION-2;
- **IN** Clause**:** SELECT column1, column2....columnN FROM table_name WHERE column_name **IN** (val-1, val-2,...val-N);
- **BETWEEN** Clause**:** SELECT column1, column2....columnN FROM table_name WHERE column_name **BETWEEN** val-1 AND val-2;
- **LIKE** Clause**:** SELECT column1, column2....columnN FROM table_name WHERE column_name **LIKE** { PATTERN };
- **ORDER BY** Clause**:** SELECT column1, column2....columnN FROM table_name WHERE CONDITION **ORDER BY** column_name {ASC|DESC};
- **GROUP BY** Clause**:** SELECT SUM(column_name) FROM table_name WHERE CONDITION **GROUP BY** column_name;
- **COUNT** Clause**:** SELECT **COUNT**(column_name) FROM table_name WHERE CONDITION;
- **HAVING** Clause**:** SELECT SUM(column_name) FROM table_name WHERE CONDITION GROUP BY column_name **HAVING** (arithematic function condition);
- **CREATE TABLE** Statement**: CREATE TABLE** table_name(column1 datatype, column2 datatype, column3 datatype, ... columnN datatype,PRIMARY KEY(one or more columns));

SQL Statements and Clauses (continued)

- **DROP TABLE** Statement**: DROP TABLE** table_name;
- **CREATE INDEX** Statement**: CREATE UNIQUE INDEX** index_name ON table_name (column1, column2,...columnN);
- **DROP INDEX** Statement **:** ALTER TABLE table_name **DROP INDEX** index_name;
- **DESC** Statement **: DESC** table_name;
- **TRUNCATE TABLE** Statement : TRUNCATE TABLE table_name;
- **ALTER TABLE** Statement **:** ALTER TABLE table_name {ADD|DROP|MODIFY} column_name {data_ype};
- **ALTER TABLE** Statement **(Rename) :** ALTER TABLE table_name **RENAME TO** new_table_name;
- **INSERT INTO** Statement **: INSERT INTO** table_name(column1, column2....columnN) VALUES (value1, value2....valueN);
- **UPDATE** Statement **: UPDATE** table_name SET column1 = value1, column2 = value2....columnN=valueN [WHERE CONDITION];
- **DELETE** Statement **: DELETE** FROM table_name WHERE {CONDITION};
- **CREATE DATABASE** Statement **: CREATE DATABASE** database_name;
- **DROP DATABASE** Statement **: DROP DATABASE** database_name;
- **USE** Statement **: USE** database_name;
- **COMMIT** Statement **: COMMIT**;
- **ROLLBACK** Statement: **ROLLBACK**;

http://docs.oracle.com/html/E13946_04/ejb3_langref.html

JAX-RS Annotations, Java Entity Classes and RESTful Web Services

Entity Classes

- Entity classes are used to create persistent entity objects
- Entity classes are **POJO** (Plain Old Java Object) classes
- Entity classes have the **@Entity** annotation
- Entity classes import `javax.persistence.Entity`
- Entity classes include getter and setter methods
- Entity classes use **annotations** above each of the variables for *keys*, *column names*, *not-null* attributes, and *relationships*

```
@Id
  @NotNull
  @Column(name = "STUDENTID")
  private Integer studentid;
  @JoinColumn(name = "COURSEID",  referencedColumnName = "COURSEID")
  @ManyToOne(optional = false)
```

Entity Class Example

Creating Entity Classes

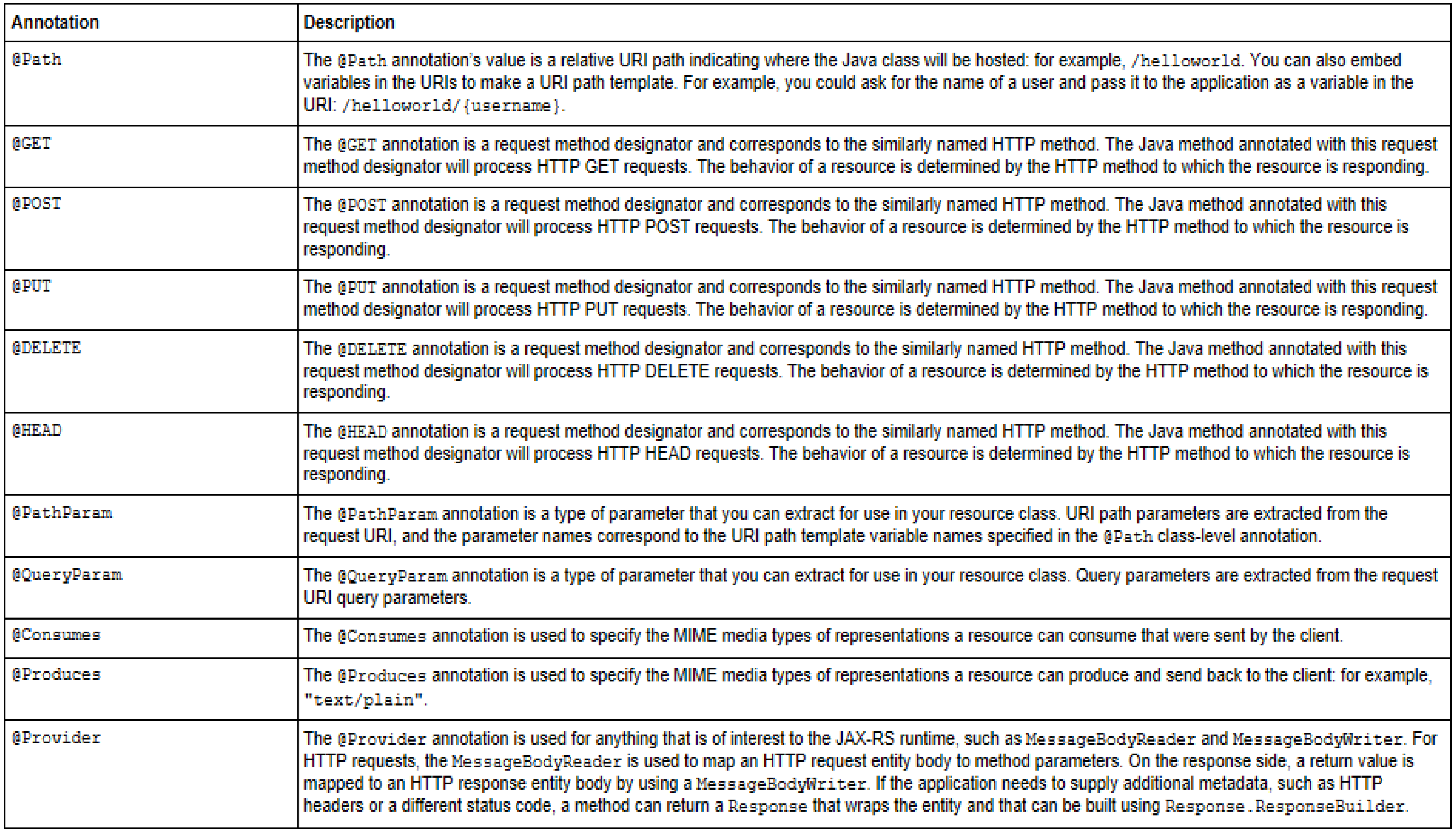
Object Serialization to XML using JAXB as part of Jersey support or JAX-RS web services

JAX-RS supports both XML and JSON.
REST WS methods can accept JSON data for JAXB annotated classes, e.g.
@Consumes({"application/xml","application/json"})
The default response is XML but to change to JSON, the client should include *accept: application/json* in the GET request

```
import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;

@Entity
@Table(name = "STUDENT")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Student.findAll", query = "SELECT s FROM Student s")
    , @NamedQuery(name = "Student.findByStudentid", query = "SELECT s FROM Student s WHERE s.studentid = :studentid")
    , @NamedQuery(name = "Student.findByStudentname", query = "SELECT s FROM Student s WHERE s.studentname =
:studentname")
    , @NamedQuery(name = "Student.findByStudpostcode", query = "SELECT s FROM Student s WHERE s.studpostcode =
:studpostcode")})
public class Student implements Serializable {
```

Summary of JAX-RS Annotations



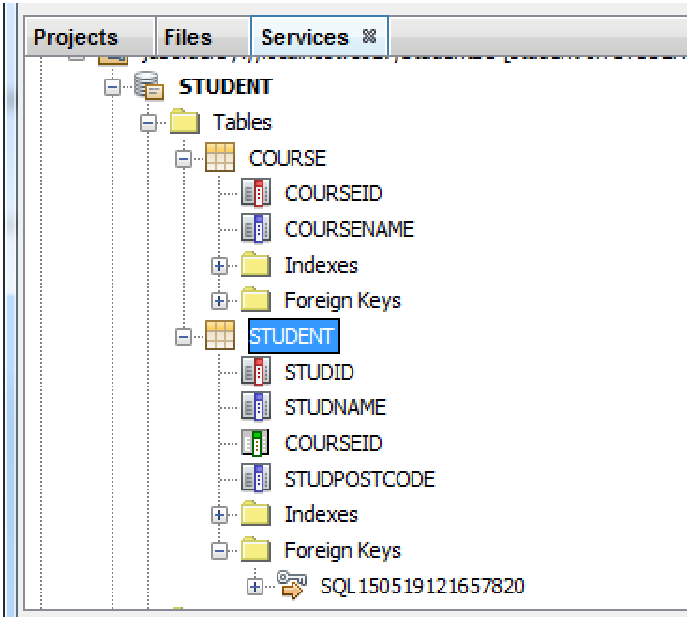
Annotation	Description
`@Path`	The `@Path` annotation's value is a relative URI path indicating where the Java class will be hosted: for example, `/helloworld`. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user and pass it to the application as a variable in the URI: `/helloworld/{username}`.
`@GET`	The `@GET` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@POST`	The `@POST` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PUT`	The `@PUT` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@DELETE`	The `@DELETE` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@HEAD`	The `@HEAD` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PathParam`	The `@PathParam` annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the `@Path` class-level annotation.
`@QueryParam`	The `@QueryParam` annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters.
`@Consumes`	The `@Consumes` annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
`@Produces`	The `@Produces` annotation is used to specify the MIME media types of representations a resource can produce and send back to the client: for example, `"text/plain"`.
`@Provider`	The `@Provider` annotation is used for anything that is of interest to the JAX-RS runtime, such as `MessageBodyReader` and `MessageBodyWriter`. For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a `MessageBodyWriter`. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity and that can be built using `Response.ResponseBuilder`.

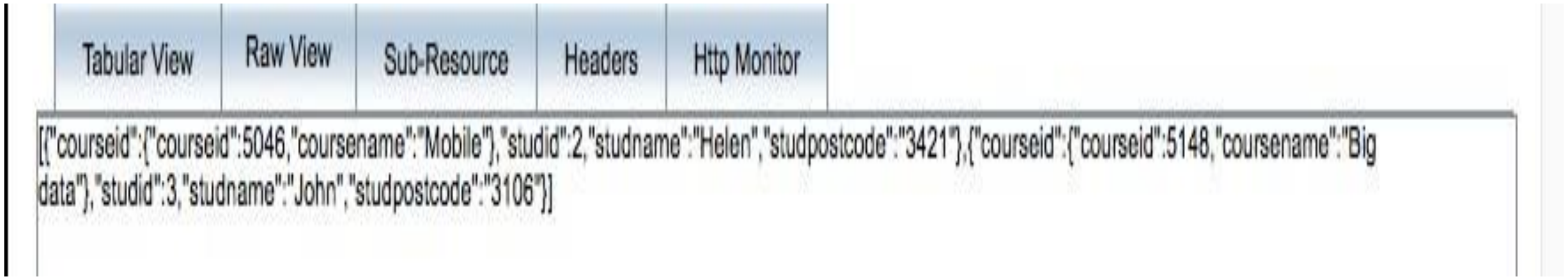
Source http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html

Tutorial Extending REST Methods

- ❑ First, create 2 tables: **Student** and **Course**
  - – 1-m relationship (courseID)

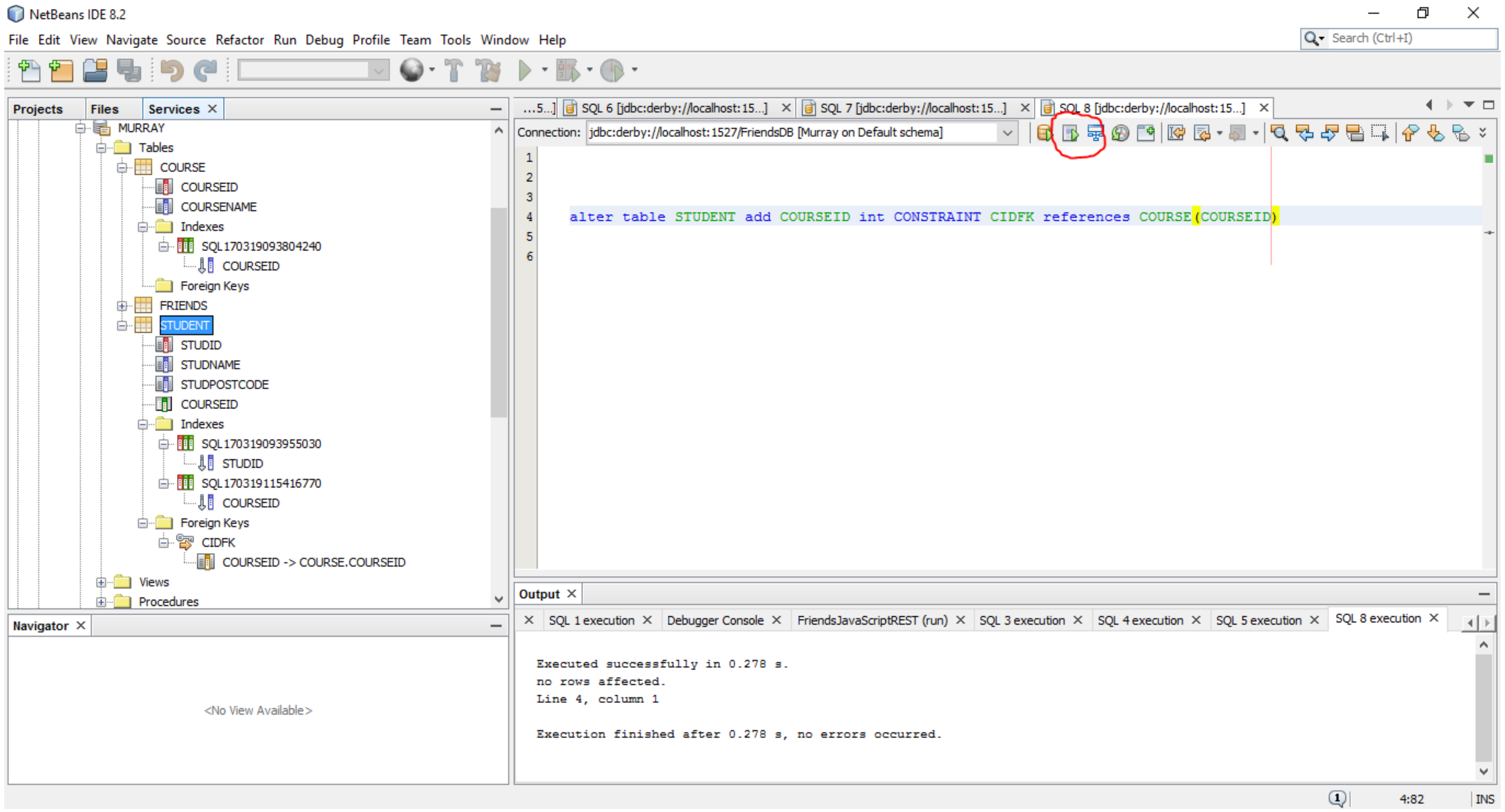
- ❑ Create entities from the database
- ❑ Create RESTful WS from entities
- ❑ Check GET all students





```
[{"courseid":{"courseid":5046,"coursename":"Mobile"},"studid":2,"studname":"Helen","studpostcode":"3421"},{"courseid":{"courseid":5148,"coursename":"Big data"},"studid":3,"studname":"John","studpostcode":"3106"}]
```

Adding Column to a Table with a Foreign Key Constraint

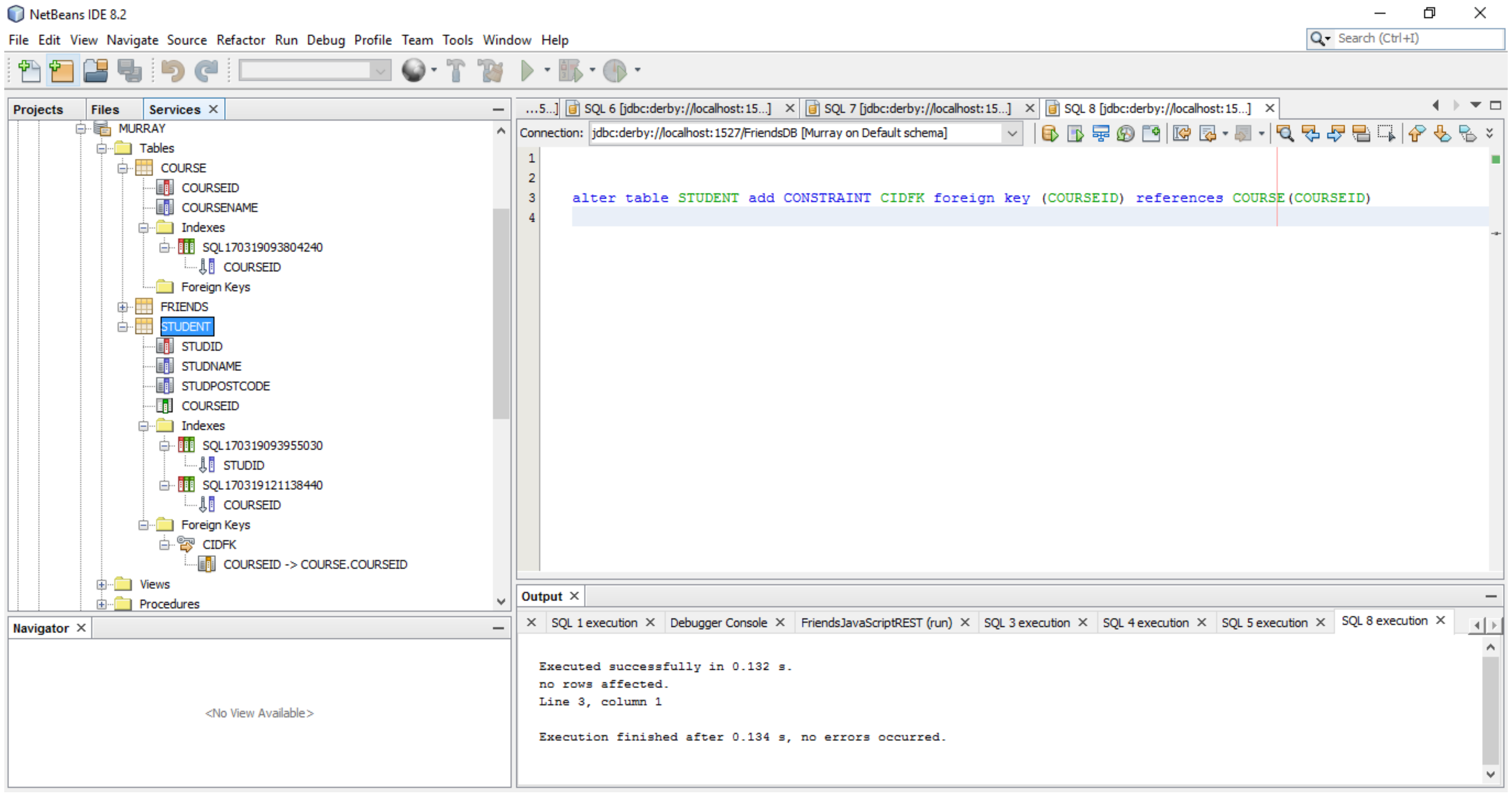


```
alter table STUDENT add COURSEID int CONSTRAINT CIDFK references COURSE(COURSEID)
```

```
Executed successfully in 0.278 s.
no rows affected.
Line 4, column 1

Execution finished after 0.278 s, no errors occurred.
```

Adding Foreign Key Constraint to existing Table Column



Connection: jdbc:derby://localhost:1527/FriendsDB [Murray on Default schema]

```
alter table STUDENT add CONSTRAINT CIDFK foreign key (COURSEID) references COURSE(COURSEID)
```

```
Executed successfully in 0.132 s.
no rows affected.
Line 3, column 1

Execution finished after 0.134 s, no errors occurred.
```

Relationships

- ❑ Check the Course Entity class for the relationship details

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "courseid")
```

"If the relationship is bidirectional, the mappedBy element must be used to specify the relationship field or property of the entity that is the owner of the relationship." (Oracle Java EE)

- ❑ Examine the Student Entity class

```
@JoinColumn(name = "COURSEID", referencedColumnName =
"COURSEID")
@ManyToOne (optional = false)
```

Optional – default is **true**. It means whether the association is optional. If set to false then a non-null relationship must always exist

Example: Course.java

```
/*
 * To change this license header, choose License Headers
in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package restClient2;

import java.io.Serializable;
import java.util.Collection;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.OneToMany;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlTransient;

/**
 *
 * @author Murray
 */
@Entity
@Table(name = "COURSE")
@XmlRootElement

@NamedQueries({
    @NamedQuery(name = "Course.findAll",
query = "SELECT c FROM Course c")
    , @NamedQuery(name = "Course.findByCourseid",
query = "SELECT c FROM Course c
WHERE c.courseid = :courseid")
    , @NamedQuery(name = "Course.findByCoursename",
query = "SELECT c FROM Course c
WHERE c.coursename = :coursename")})

public class Course implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "COURSEID")
    private Integer courseid;
    @Size(max = 7)
    @Column(name = "COURSENAME")
    private String coursename;

    @OneToMany(mappedBy = "courseid")
    private Collection<Student> studentCollection;

    public Course() {
    }

    public Course(Integer courseid) {
        this.courseid = courseid;
    }

    public Integer getCourseid() {
        return courseid;
    }

    public void setCourseid(Integer courseid) {
        this.courseid = courseid;
    }

    public String getCoursename() {
        return coursename;
    }

    public void setCourseid(Integer courseid) {
        this.courseid = courseid;
    }

    public String getCoursename() {
        return coursename;
    }

    public void setCoursename(String coursename) {
        this.coursename = coursename;
    }

    @XmlTransient
    public Collection<Student> getStudentCollection() {
        return studentCollection;
    }

    public void setStudentCollection
(Collection<Student> studentCollection) {
        this.studentCollection = studentCollection;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (courseid != null ? courseid.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the
case the id fields are not set
        if (!(object instanceof Course)) {
            return false;
        }
        Course other = (Course) object;
        if ((this.courseid == null && other.courseid != null) ||
(this.courseid != null &&
!this.courseid.equals(other.courseid))) {
            return false;
        }
        return true;
    }
    @Override
    public String toString() {
        return "restClient2.Course[ courseid=" + courseid + " ]";
    }
}
```

Example: Student.java

```java
/*
 * To change this license header, choose License Headers
in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.*/

package restClient2;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
/**
 *
 * @author Murray
 */
@Entity
@Table(name = "STUDENT")
@XmlRootElement

@NamedQueries({
    @NamedQuery(name = "Student.findAll",
query = "SELECT s FROM Student s")
    , @NamedQuery(name = "Student.findByStudid",
query = "SELECT s FROM Student s
WHERE s.studid = :studid")
    , @NamedQuery(name = "Student.findByStudname",
query = "SELECT s FROM Student s
WHERE s.studname = :studname")
    , @NamedQuery(name = "Student.findByStudpostcode",
query = "SELECT s FROM Student s
WHERE s.studpostcode = :studpostcode")})

public class Student implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "STUDID")
    private Integer studid;
    @Size(max = 20)
    @Column(name = "STUDNAME")
    private String studname;
    @Column(name = "STUDPOSTCODE")
    private Integer studpostcode;

    @JoinColumn(name = "COURSEID",
referencedColumnName = "COURSEID")

    @ManyToOne
    private Course courseid;

    public Student() {
    }

    public Student(Integer studid) {
        this.studid = studid;
    }

    public Integer getStudid() {
        return studid;
    }

    public void setStudid(Integer studid) {
        this.studid = studid;
    }

    public String getStudname() {
        return studname;
    }

    public void setStudname(String studname) {
        this.studname = studname;
    }

    public Integer getStudpostcode() {
        return studpostcode;
    }

    public void setStudpostcode(Integer studpostcode) {
        this.studpostcode = studpostcode;
    }

    public Course getCourseid() {
        return courseid;
    }

    public void setCourseid(Course courseid) {
        this.courseid = courseid;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (studid != null ? studid.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in
the case the id fields are not set
        if (!(object instanceof Student)) {
            return false;
        }
        Student other = (Student) object;
        if ((this.studid == null && other.studid != null) ||
(this.studid != null && !this.studid.equals(other.studid))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "restClient2.Student[ studid=" + studid + " ]";
    }
}
```

Example: CourseFacadeREST.java

```
/* To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package restClient2.service;

import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import restClient2.Course;
/**
 * @author Murray
 */
@Stateless
@Path("restclient2.course")
public class CourseFacadeREST extends AbstractFacade<Course> {

    @PersistenceContext (unitName = "FriendsDBPU")
    private EntityManager em;

    public CourseFacadeREST() {
        super(Course.class);
    }
 @POST
    @Override
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public void create(Course entity) {
        super.create(entity);
    }
    @PUT
    @Path("{id}")
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public void edit(@PathParam("id") Integer id, Course entity) {
        super.edit(entity);
    }
    @DELETE
    @Path("{id}")
    public void remove(@PathParam("id") Integer id) {
        super.remove(super.find(id));
    }
    @GET
    @Path("{id}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Course find(@PathParam("id") Integer id) {
        return super.find(id);
    }
    @GET
    @Override
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public List<Course> findAll() {
        return super.findAll();
    }
    @GET
    @Path("{from}/{to}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public List<Course> findRange(@PathParam("from") Integer from, @PathParam("to")
Integer to) {
        return super.findRange(new int[]{from, to});
    }
    @GET
    @Path("count")
    @Produces(MediaType.TEXT_PLAIN)
    public String countREST() {
        return String.valueOf(super.count());
    }
    @Override
    protected EntityManager getEntityManager() {
        return em;
    }
}
```

Example: StudentFacadeREST.java

```
/* To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package restclient.service;

import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import restclient.Student;
/**
 * @author Murray
 */
@Stateless
@Path("restclient.student")
public class StudentFacadeREST extends AbstractFacade<Student> {

    @PersistenceContext(unitName = "FriendsDBPU")
    private EntityManager em;

    public StudentFacadeREST() {
        super(Student.class);
    }
    @POST
    @Override
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public void create(Student entity) {
        super.create(entity);
    }
    @PUT
    @Path("{id}")
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public void edit(@PathParam("id") Integer id, Student entity) {
        super.edit(entity);
    }
    @DELETE
    @Path("{id}")
    public void remove(@PathParam("id") Integer id) {
        super.remove(super.find(id));
    }
    @GET
    @Path("{id}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Student find(@PathParam("id") Integer id) {
        return super.find(id);
    }
    @GET
    @Override
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public List<Student> findAll() {
        return super.findAll();
    }
    @GET
    @Path("{from}/{to}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public List<Student> findRange(@PathParam("from") Integer from, @PathParam("to")
Integer to) {
        return super.findRange(new int[]{from, to});
    }
    @GET
    @Path("count")
    @Produces(MediaType.TEXT_PLAIN)
    public String countREST() {
        return String.valueOf(super.count());
    }
…

    }
   @Override
    protected EntityManager getEntityManager() {
        return em;
    }
}
```

Named Queries

NamedQuery

Examine the Student Entity class

```
@Entity
@Table(name = "STUDENT")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Students.findAll", query = "SELECT s
FROM Students s"),
    @NamedQuery(name = "Students.findByStudid", query =
"SELECT s FROM Students s WHERE s.studid = :studid"),
    @NamedQuery(name = "Students.findByStudname", query =
"SELECT s FROM Students s WHERE s.studname = :studname"),
    @NamedQuery(name = "Students.findByStudpostcode", query =
"SELECT s FROM Students s WHERE s.studpostcode
=   :studpostcode")})
```

Adding a New REST Method

You can create GET methods for each attribute using existing NamedQueries. Example:

StudentFacadeREST.java

```
@Stateless
@Path("student.student")
public class StudentFacadeREST extends AbstractFacade<Student> {
…
@Path("findByStudname/{studname}")
   @Produces({"application/json"})
   public List<Student> findByStudname(@PathParam("studname") String studname) {
      Query query = em.createNamedQuery("Student.findByStudname");
//em is EntityManager
      query.setParameter("studname", studname);
      return query.getResultList();
   }
```

The entity class Student.java

```
@NamedQuery(name = "Student.findByStudname", query = "SELECT s FROM
Student s WHERE s.studname = :studname"),
```

Adding New Queries

- ❑ **Two options**

- **Static** queries (define the query in the entity class and invoke from the REST façade class)

- **Dynamic** queries (define the query in REST façade class only)

Option 1 – Static Query

- Write the NamedQuery in the Entity class
- The code below creates 3 different NamedQueries

```
@NamedQuery(name = "Student.findByCourse", query = "SELECT s
FROM Student s WHERE s.courseid.courseid = :courseid"),

@NamedQuery(name = "Student.findByUpperStudname", query =
"SELECT s FROM Student s WHERE UPPER(s.studname) =
UPPER(:studname)"),

@NamedQuery(name = "Student.findByLikePostcode", query = "SELECT
s FROM Student s WHERE s.studpostcode LIKE '3%'"),
```

- Then you will create a new REST method based on the NamedQuery

(Note: If you make changes to the entities, you will have to recreate RESTful WS from entities)

REST Methods

```
@GET
    @Path("findByLikePostcode")
    @Produces({"application/json"})
    public List<Student> findByLikePostCode() {
        Query query = em.createNamedQuery("Student.findByLikePostcode");
            return query.getResultList();
    }
@GET
    @Path("findByCourse/{courseid}")
    @Produces({"application/json"})
    public List<Student> findByCourse(@PathParam("courseid") Integer courseid) {
        Query query = em.createNamedQuery("Student.findByCourse");
        query.setParameter("courseid", courseid);
        return query.getResultList();
    }
@GET
@Path("findByUpperStudname/{studname}")
    @Produces({"application/json"})
    public List<Student> findByUpperStudname(@PathParam("studname") String
studname) {
        Query query = em.createNamedQuery("Student.findByUpperStudname");
        query.setParameter("studname", studname);
        return query.getResultList();     }
```

Option 2 – Dynamic Query

We can create the queries directly in the REST method instead of defining it in the Entity Class by calling createQuery()

```
@GET
   @Path("findByLikePostcode")
   @Produces({"application/json"})
   public List<Student> findByLikePostcode() {
      TypedQuery<Student> query = em.createQuery("SELECT s FROM Student s WHERE
s.studpostcode LIKE '3%'", Student.class);
      return query.getResultList();
   }
 @GET
   @Path("findByCourse/{courseid}")
   @Produces({"application/json"})
   public List<Student> findByCourse(@PathParam("courseid") Integer courseid) {
      TypedQuery<Student> q = em.createQuery("SELECT s FROM Student s WHERE
s.courseid.courseid = :courseid", Student.class);
      q.setParameter("courseid", courseid);
      return q.getResultList();
   }
```

Composite AND and JOIN Queries

AND Queries

- Static query – first write a NamedQuery in the entity class

```
@NamedQuery(name = "Student.findByNameANDCode",
query = "SELECT s FROM Student s WHERE
UPPER(s.studname) = UPPER(:studname) AND
s.studpostcode = :studpostcode"),
```

- Then write its REST method
- Path includes two parameters with a certain order

```
@GET
    @Path("findByNameANDCode/{studname}/{studpostcode}")
    @Produces({"application/json"})
    public List<Student> findByNameANDCode(@PathParam("studname")
String studname, @PathParam("studpostcode") String studpostcode) {
        Query query =
em.createNamedQuery("Student.findByNameANDCode");
        query.setParameter("studname", studname);
        query.setParameter("studpostcode", studpostcode);
        return query.getResultList();
```

Implicit JOIN

- ❑ Student and Course tables have 1-m relationship
- ❑ You can make queries that involves attributes from both tables
- ❑ See how in the example below you are accessing 'coursename' from the Student s based on the join attribute (`courseid`) using an *implicit* join.

```
@GET
     @Path("findByCourseNsme/{coursename}")
     @Produces({"application/json"})
     public List<Student>
findByCourseName(@PathParam("coursename") String coursename)
{
          TypedQuery<Student> q = em.createQuery("SELECT s FROM
Student s WHERE s.courseid.coursename = :coursename",
Student.class);
          q.setParameter("coursename     ", coursename);
          return q.getResultList();
     }
```

Explicit JOIN

It requires writing a query that **explicitly** uses the ‘JOIN’ clause in the query

[INNER] JOIN … [ON] … [WHERE] …

By default [INNER] join is used in JPQL

LEFT [OUTER] JOIN is possible for outer joins

[INNER] JOIN

As discussed above, range variables represent iteration over all the database objects of a specified entity type. JPQL provides an additional type of identification variable, a join variable, which represent a more limited iteration over specified collections of objects. The following query uses one range variable and one join variable:

```
SELECT c1, c2 FROM Country c1 INNER JOIN c1.neighbors c2
```

LEFT [OUTER] JOIN

To understand the purpose of OUTER JOIN, consider the following INNER JOIN query that retrieves pairs of (country name, capital name):

```
SELECT c.name, p.name FROM Country c JOIN c.capital p
```

The behavior of OUTER JOIN is different, as demonstrated by the following query variant:

```
SELECT c, p.name FROM Country c LEFT OUTER JOIN c.capital p
```

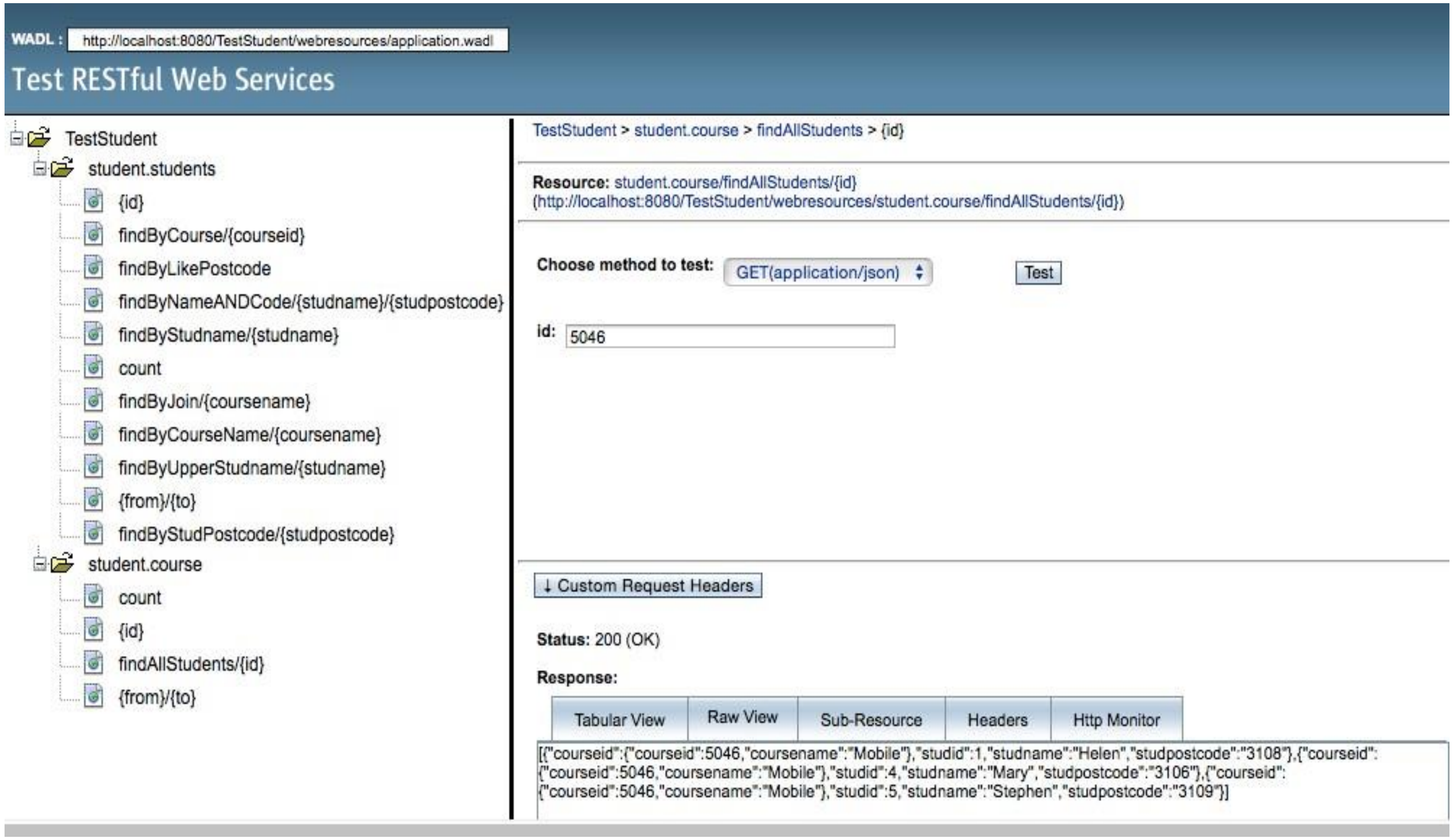
Source http://www.objectdb.com/java/jpa/query/jpql/from

Collection

- ❑ Course has a many-to-one relationship with Student. If you check its entity class, there is a method called getStudentsCollection():
  - *public Collection<Students> getStudentsCollection()*
- ❑ To get all the students based on a defined predicate/condition
- ❑ An example:

```
@GET
    @Path("**findAllStudents**/{id}")
    @Produces({"application/json"})
    public Collection<Student> findAllstudents(@PathParam("id")
Integer id) {
   Query query = em.createQuery("Select c from Course c WHERE
c.courseid = :id", Course.class);
        query.setParameter("id", id);
        Course singleResult = (Course) query.getSingleResult();
        return singleResult.getStudentsCollection();
    }
```

Test Web Services



WADL : http://localhost:8080/TestStudent/webresources/application.wadl

Test RESTful Web Services

TestStudent > student.course > findAllStudents > {id}

Resource: student.course/findAllStudents/{id}
(http://localhost:8080/TestStudent/webresources/student.course/findAllStudents/{id})

Choose method to test: GET(application/json) Test

id: 5046

Status: 200 (OK)

Response:

```
[{"courseid":{"courseid":5046,"coursename":"Mobile"},"studid":1,"studname":"Helen","studpostcode":"3108"},{"courseid":
{"courseid":5046,"coursename":"Mobile"},"studid":4,"studname":"Mary","studpostcode":"3106"},{"courseid":
{"courseid":5046,"coursename":"Mobile"},"studid":5,"studname":"Stephen","studpostcode":"3109"}]
```

JPA and JPQL Reference List

- http://docs.oracle.com/html/E13946_04/ejb3_langref.html
- http://docs.oracle.com/javaee/6/tutorial/doc/bnbtg.html
- http://docs.oracle.com/javaee/6/tutorial/doc/gkknj.html
- https://openjpa.apache.org/builds/1.2.3/apache-openjpa/docs/jpa_overview_query.html

Collaboration during the development phase and peer code review

The following book by Cohen J. (2013) describes 5 different approaches to peer code review (see Moodle for excerpt):

1. Formal reviews
2. Over the shoulder
3. Email pass-around
4. Tool assisted
5. Pair programming

Each of these have their strengths and weaknesses, but the conclusion is that:

Any kind of code review is better than nothing!

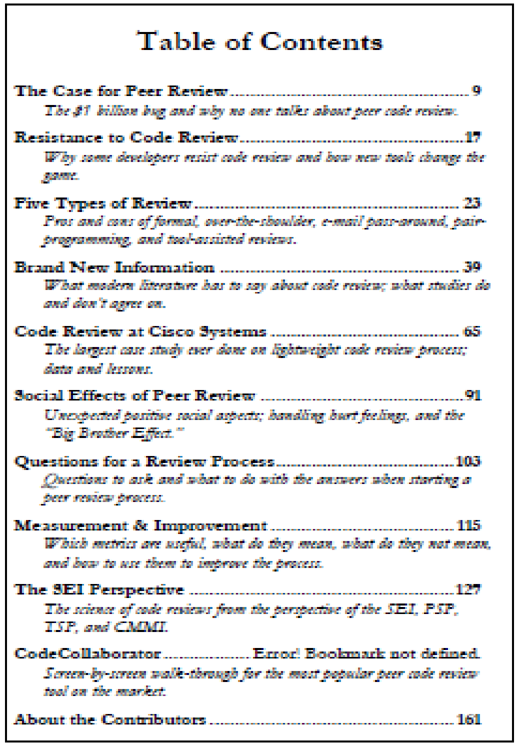


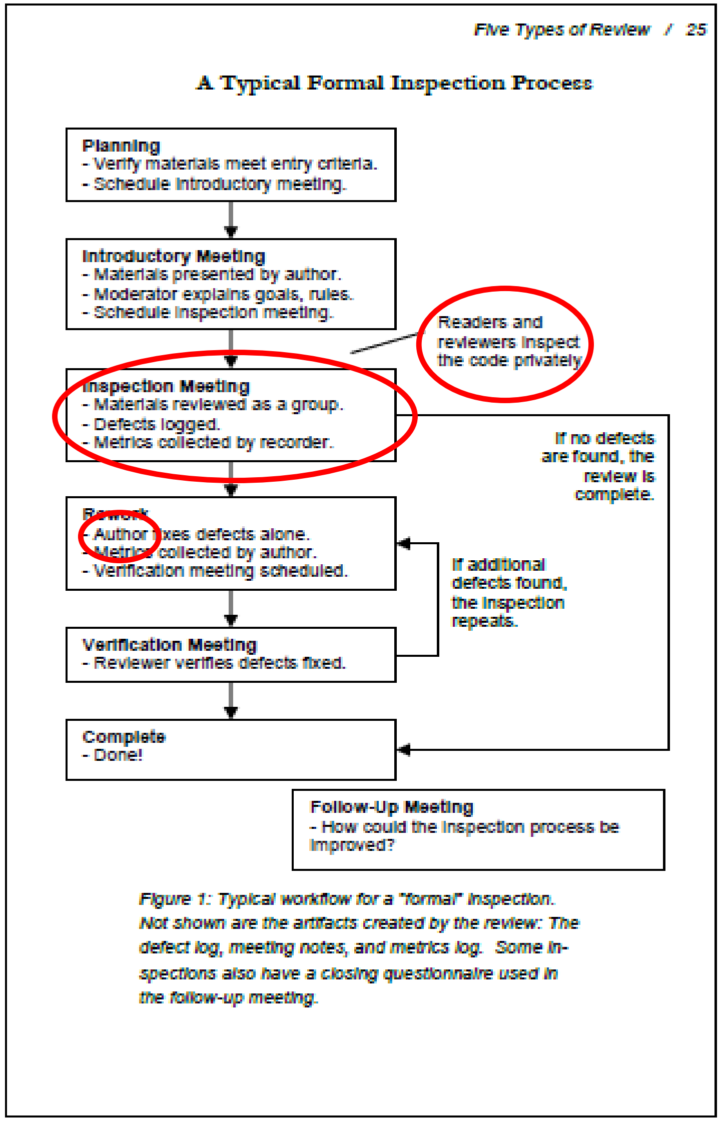
Table of Contents

Formal peer code review

(Fagan/Gilb methods)

The software development process is a typical application of Fagan Inspection; software development process is a series of operations which will deliver a certain end product and consists of operations like requirements definition, design, coding up to testing and maintenance. As the costs to remedy a defect are up to 10-100 times less in the early operations compared to fixing a defect in the maintenance phase it is essential to find defects as close to the point of insertion as possible. This is done by inspecting the output of each operation and comparing that to the output requirements, or exit-criteria of that operation. *-Wikipedia*

We will use simplified version of this process or code walk-throughs and demonstrations for the group Assignments!


Five Types of Review / 25

A Typical Formal Inspection Process

Planning
- Verify materials meet entry criteria.
- Schedule introductory meeting.
Introductory Meeting
- Materials presented by author.
- Moderator explains goals, rules.
- Schedule inspection meeting.
Readers and reviewers inspect the code privately
Inspection Meeting
- Materials reviewed as a group.
- Defects logged.
- Metrics collected by recorder.
If no defects are found, the review is complete.
Rework
- Author fixes defects alone.
- Metrics collected by author.
- Verification meeting scheduled.
If additional defects found, the inspection repeats.
Verification Meeting
- Reviewer verifies defects fixed.
Complete
- Done!
Follow-Up Meeting
- How could the inspection process be improved?

Figure 1: Typical workflow for a "formal" inspection. Not shown are the artifacts created by the review: The defect log, meeting notes, and metrics log. Some inspections also have a closing questionnaire used in the follow-up meeting.