CT5106

JPA & EJB's

Entities

- Java Persistence API requires that you identify the classes that you will store in a database.
- The API uses the term entity to define classes that it will map to a relational database. You identify persistable entities and define their relationships using annotations.
- The Java compiler recognizes and uses annotations to save your work. Using annotations, the compiler can generate additional classes for you and perform compile time error checking.

Table Name Defaults

- Every entity has a name. By default, the entity name is the entity's class name.
- The Entity annotation has a name attribute that allows you to explicitly specify a name.
- You will use the entity name in queries

```
@Entity
@Table(name = "customer")
```

Column name defaults

- By default, persistence provider implementations use an entity's field or property names as the column names in the entity table
- You can, however, override the default using the Column annotation and its name element. If you prefer SURNAME instead of LASTNAME, you can annotate the lastName property like this:

```
@Size(max = 255)
@Column(name = "Address")
private String address;
```

Entity Declarations

- Entity classes become tables in a relational database. Entity instances map into rows in one or more tables.
- The following code sample begins the definition of a baseball Player class. Annotations in your code begin with the @ symbol.
- The Java Persistence API implementation *can* create a table for the Player entity in your relational database.
- By default, the table name corresponds to the unqualified class name. In this case, a PLAYER table will represent Player entities.

```
@Entity
public class Player implements Serializable
{
 ……
```

Primary Keys

- All entities must have a primary key. Keys can be a single unique field or a combination of fields.
- Identify single field keys with the Id annotation.
- Key fields must have one of the following types:
  - primitive type (like int, long, etc)
  - wrappers for primitive types (Integer, Long, etc)
  - java.lang.String
  - java.util.Date
  - java.sql.Date

The reference implementation will generate a key automatically if you add the @GeneratedValue annotation to the primary key:

Sequencing

- In JPA the object id is defined through the @Id annotation and should correspond to the primary key of the object's table.
- An object id can either be a natural id (e.g. email, name, PRSI) or a generated id.
  - The main issue with natural ids is that almost anything can change eventually. Natural ids can also make querying, foreign keys and indexing less efficient in the database.
- A sequence number in JPA is a sequential id generated by the JPA implementation and automatically assigned to new objects.
  - The benefits of using sequence numbers are that they are guaranteed to be unique, allow all other data of the object to change, are efficient values for querying and indexes, and can be efficiently assigned.
- In JPA an @Id can be easily assigned a generated sequence number through the @GeneratedValue annotation

(unless you just want to generate them yourself – then just don't use @GeneratedValue)

@GeneratedValue(strategy = GenerationType.IDENTITY)

- popular method
- Not necessarily supported by all databases

  - The database controls the ID generation, JPA does not act on the id at all. Thus in order to retrieve the id from the databse an entity needs to be persisted first, and after the transaction commits, a query is executed by the entity manager to retrieve the generated id for the specific entity.

MySQL Identity column use

```
CREATE TABLE name (
Id int NOT NULL AUTO_INCREMENT,
Firstname varchar(30) DEFAULT NULL,
PRIMARY KEY (Id)) ;
Query OK, 0 rows affected
-- Inserting values into table.
INSERT INTO Name(FirstName) VALUES ('John');
INSERT INTO Name(FirstName) VALUES ('Mary');
INSERT INTO Name(FirstName) VALUES ('Peter');
select * from name;
```

ID	FirstName
1	John
2	Mary
3	Peter

SQL Server Identity use

```
CREATE TABLE Name(
    ID INT IDENTITY NOT NULL PRIMARY KEY,
    FirstName VARCHAR(40) NOT NULL
);
-- Inserting values into table.
INSERT INTO Name(FirstName) VALUES ('John');
INSERT INTO Name(FirstName) VALUES ('Mary');
INSERT INTO Name(FirstName) VALUES ('Peter');
SELECT * FROM dbo.Name;
```

ID	FirstName
1	John
2	Mary
3	Peter

Category entity uses IDENTITY

```
-- -----------------------------------------------------
-- Table `affablebean`.`category`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `affablebean`.`category` ;

CREATE  TABLE IF NOT EXISTS `affablebean`.`category` (
  `id` TINYINT UNSIGNED NOT NULL AUTO_INCREMENT ,
  `name` VARCHAR(45) NOT NULL ,
  PRIMARY KEY (`id`) )
ENGINE = InnoDB
COMMENT = 'contains product categories, e.g., dairy, meats, etc.';
```

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Basic(optional = false)
@Column(name = "id")
private Short id;
```

GenerationType.SEQUENCE

- Used with sequence generator (created in database based on annotation, e.g.:
  - @SequenceGenerator(name = Car.CAR_SEQUENCE_NAME, sequenceName = Car.CAR_SEQUENCE_NAME, initialValue = 10, allocationSize = 53)
  - @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = CAR_SEQUENCE_NAME)
  - allocationSize = # of id's JPA keeps in a cache to used
  - Not all databases support sequences – MySQL doesn't for example – closest thing in MySQL is AUTO INCREMENT (you can set the increment value)

Create sequence in SQL Server

```
CREATE SEQUENCE Sequence_Name
START WITH Initial_Value
INCREMENT BY Increment_Value
MINVALUE Minimum_Value
MAXVALUE Maximum_Value
CYCLE|NOCYCLE;
```

- **Sequence_Name** – This specifies the name of the sequence.
- **Initial_Value** – This specifies the starting value from where the sequence should start.
- **Increment_Value** – This specifies the value by which the sequence will increment by itself. This can be valued positively or negatively.
- **Minimum_Value** – This specifies the minimum value of the sequence.
- **Maximum_Value** – This specifies the maximum value of the sequence.
- **Cycle** – When the sequence reaches its Maximum_Value, it starts again from the beginning.
- **Nocycle** – An exception will be thrown if the sequence exceeds the Maximum_Value.

Use sequence in SQL Server

```
CREATE TABLE CUSTOMERS (
   ID INT,
   NAME VARCHAR (20) NOT NULL,
   AGE INT NOT NULL,
   ADDRESS CHAR (25),
   SALARY DECIMAL (18, 2),
);
```

```
INSERT INTO CUSTOMERS VALUES
(NULL, 'Ramesh', 32, 'Ahmedabad', 2000.00),
(NULL, 'Khilan', 25, 'Delhi', 1500.00),
(NULL, 'Kaushik', 23, 'Kota', 2000.00),
(NULL, 'Chaitali', 25, 'Mumbai', 6500.00),
(NULL, 'Hardik', 27, 'Bhopal', 8500.00),
(NULL, 'Komal', 22, 'Hyderabad', 4500.00),
(NULL, 'Muffy', 24, 'Indore', 10000.00 );
```

ID	NAME	AGE	ADDRESS	SALARY
NULL	Ramesh	32	Ahmedabad	2000.00
NULL	Khilan	25	Delhi	1500.00
NULL	Kaushik	23	Kota	2000.00
NULL	Chaitali	25	Mumbai	6500.00
NULL	Hardik	27	Bhopal	8500.00

Use sequence in SQL Server

```
CREATE SEQUENCE My_Sequence AS INT
START WITH 1
INCREMENT BY 1
MINVALUE 1
MAXVALUE 7
CYCLE;
```

```
UPDATE CUSTOMERS SET ID = NEXT VALUE FOR my_Sequence;
```

```
SELECT * FROM CUSTOMERS;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00

GenerationType.TABLE

- A table is used to store the id's e.g.

```
...
@Entity
public class Employee {
    @Id
    @TableGenerator(name="TABLE_GEN", table="SEQUENCE_TABLE", pkColumnName="SEQ_NAME",
        valueColumnName="SEQ_COUNT", pkColumnValue="EMP_SEQ")
    @GeneratedValue(strategy=GenerationType.TABLE, generator="TABLE_GEN")
    private long id;
    ...
}
```

SEQUENCE_TABLE	
SEQ_NAME	***SEQ_COUNT***
EMP_SEQ	123
PROJ_SEQ	550

GenerationType.AUTO

- With the Auto approach any strategy can be used by JPA.
- JPA will choose.
  - Picks the strategy that is preferred by the used database platform.
  - The preferred strategies are IDENTITY for MySQL, SQLite and MsSQL (SQL Server) and SEQUENCE for Oracle and PostgreSQL.

Using Table

- EclipseLink (JPA provider) apparently prefers the use of Table when Auto is chosen
- In this case, the sequence table would need to be generated by the JPA provider
- This means that you would need to run the application in ‘create tables’ mode at least once

@Basic(optional = false)

- JPA support various Java data types as persistable fields of an entity, often known as the basic types.
- **A basic type maps directly to a column in the database.** These include Java primitives and their wrapper classes, *String*, *java.math.BigInteger* and *java.math.BigDecimal*, and various available date-time classes

- false => cannot be null

@Transient

Always explicitly mark properties or fields that should not be persisted.
Use the annotation Transient for marking transient properties.

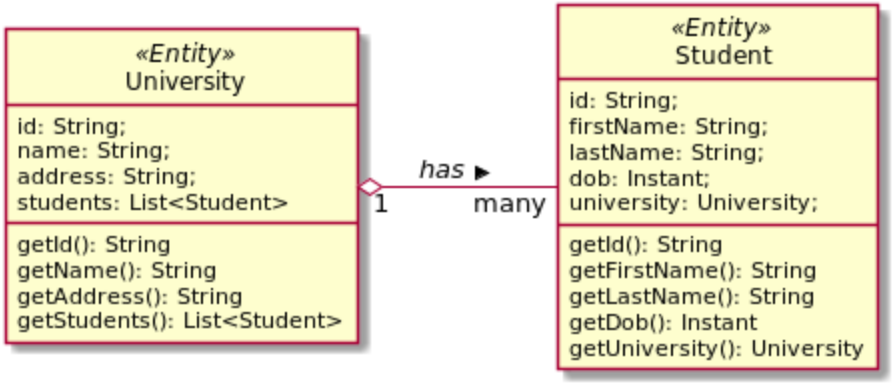
```
/**
* Returns the last words spoken by this
player.
* We don't want to persist that!
*/
@Transient
public String getLastSpokenWords(){
return lastSpokenWords;
}
```

Cascading changes to related entities

- In a JPA entity relationship, the CascadeType. ALL annotation specifies that all operations (persist, merge, remove, refresh, and detach) that are performed on the parent entity should be cascaded to the child entity

Fetching related entities

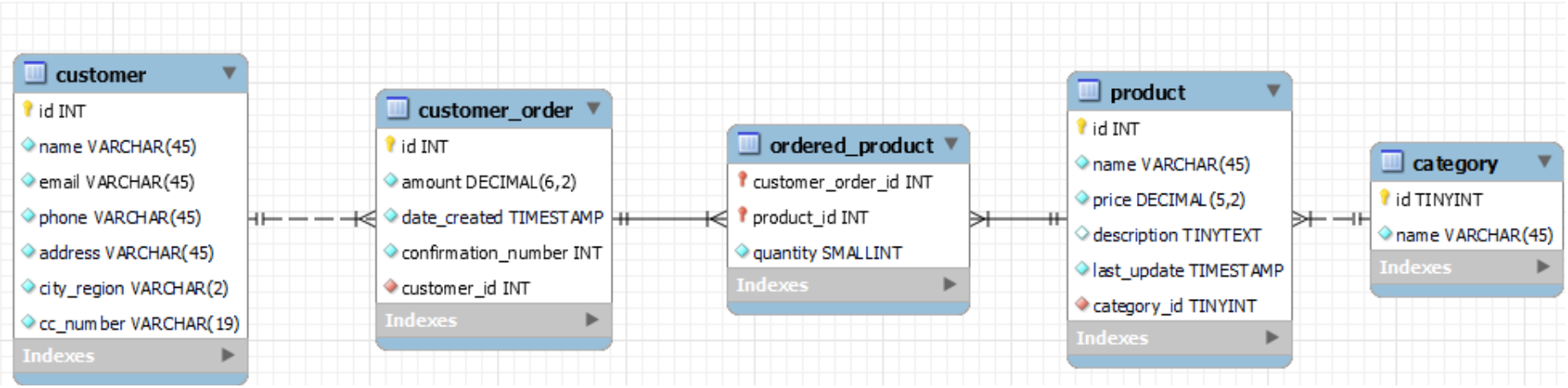
- Dictates when you load related entities
  - E.g. fetch productCollection when you load Category (EAGER) , or only when you call getProductCollection() (LAZY)




```
@Entity
public class University {
   @Id
   private String id;
   private String name;
   private String address;
   @OneToMany(fetch = FetchType.EAGER)
   private List<Student> students;
   // etc.
}
```

```
@Entity
public class University {
   @Id
   private String id;
   private String name;
   private String address;
   @OneToMany(fetch = LAZY)
   private List<Student> students;
   // etc.
}
```

example – online shop – AffableBean tutorial



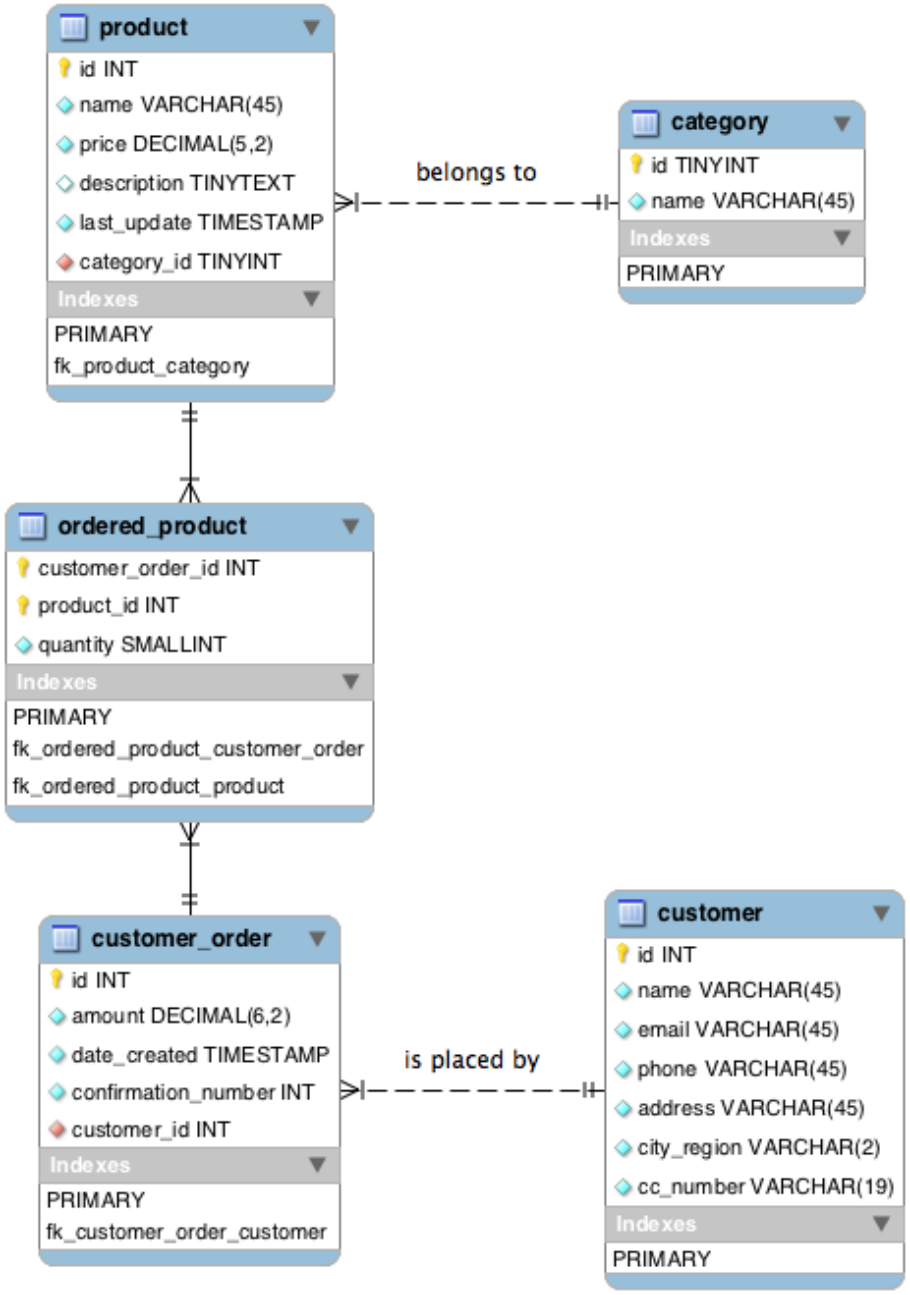
This is a MySQL database created using the following SQL:

1. createSchema.sql
2. createData.sql

https://netbeans.apache.org/kb/docs/javaee/ecommerce/intro.html

Data Model

- Basic set of entities needed
- Note the need for the ordered-product entity
  - Serves the same purpose as a line in an order
  - Represents the many-to-many relationship between product and customer_order
  - Note it has a compound primary key
- Note the multiplicity of the relationships
  - Note also the direction(s) of the relationships
- Note the need for foreign keys



customer table

- Primary key ‘id’

```
-- -----------------------------------------------------
-- Table `affablebean`.`customer`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `affablebean`.`customer` ;

CREATE  TABLE IF NOT EXISTS `affablebean`.`customer` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
  `name` VARCHAR(45) NOT NULL ,
  `email` VARCHAR(45) NOT NULL ,
  `phone` VARCHAR(45) NOT NULL ,
  `address` VARCHAR(45) NOT NULL ,
  `city_region` VARCHAR(2) NOT NULL ,
  `cc_number` VARCHAR(19) NOT NULL ,
  PRIMARY KEY (`id`) )
ENGINE = InnoDB
COMMENT = 'maintains customer details';
```

customer_order table

- Foreign key refers to PK of customer

```
-- -----------------------------------------------------
-- Table `affablebean`.`customer_order`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `affablebean`.`customer_order` ;

CREATE  TABLE IF NOT EXISTS `affablebean`.`customer_order` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
  `amount` DECIMAL(6,2) NOT NULL ,
  `date_created` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
  `confirmation_number` INT UNSIGNED NOT NULL ,
  `customer_id` INT UNSIGNED NOT NULL ,
  PRIMARY KEY (`id`) ,
  INDEX `fk_customer_order_customer` (`customer_id` ASC) ,
  CONSTRAINT `fk_customer_order_customer`
    FOREIGN KEY (`customer_id` )
    REFERENCES `affablebean`.`customer` (`id` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB
COMMENT = 'maintains customer order details';
```

category table

- Only used by product

```
-- -----------------------------------------------------
-- Table `affablebean`.`category`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `affablebean`.`category` ;

CREATE  TABLE IF NOT EXISTS `affablebean`.`category` (
  `id` TINYINT UNSIGNED NOT NULL AUTO_INCREMENT ,
  `name` VARCHAR(45) NOT NULL ,
  PRIMARY KEY (`id`) )
ENGINE = InnoDB
COMMENT = 'contains product categories, e.g., dairy, meats, etc.';
```

Category entity

```
@Entity
@Table(name = "category")
@NamedQueries({
    @NamedQuery(name = "Category.findAll", query = "SELECT c FROM Category c"),
    @NamedQuery(name = "Category.findById", query = "SELECT c FROM Category c WHERE c.id = :id"),
    @NamedQuery(name = "Category.findByName", query = "SELECT c FROM Category c WHERE c.name = :name")})
public class Category implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id")
    private Short id;
    @Basic(optional = false)
    @Column(name = "name")
    private String name;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "category")
    private Collection<Product> productCollection;
```

Named queries pre-defined, and can be run by entity manager on request

Database maintains an auto-incremented column for the id values, and this is inserted into the entity by the entity manager when persisted

Makes relationship readable in both directions

CascadeType.ALL: all operations (persist, remove, merge,..) are propagated to the children – i.e. remove Category => remove Product

Cascading dangerous – deleting parent can delete all children

product

```
-- -----------------------------------------------------
-- Table `affablebean`.`product`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `affablebean`.`product` ;

CREATE  TABLE IF NOT EXISTS `affablebean`.`product` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT ,
  `name` VARCHAR(45) NOT NULL ,
  `price` DECIMAL(5,2) NOT NULL ,
  `description` TINYTEXT NULL ,

  `last_update` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP ,
  `category_id` TINYINT UNSIGNED NOT NULL ,
  PRIMARY KEY (`id`) ,
  INDEX `fk_product_category` (`category_id` ASC) ,
  CONSTRAINT `fk_product_category`
    FOREIGN KEY (`category_id` )
    REFERENCES `affablebean`.`category` (`id` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB
COMMENT = 'contains product details';
```

Product

- Many Products to one Category
  - @JoinColumn
    - Defines the physical mapping on the owning side of the one-to-many relationship (here with category) which is usually defined on the *many* side
    - Name: the column name in this (Product) table which links to the 'referencedColumName' in the referenced table (Category)
- One Product to many OrderProducts
  - mappedBy tells us the name of the field on the owning side of the OneToMany relationships (OrderedProduct) which field is used to reference the Product

```
@JoinColumn(name = "category_id", referencedColumnName = "id")
@ManyToOne(optional = false)
private Category category;
@OneToMany(cascade = CascadeType.ALL, mappedBy = "product")
private Collection<OrderedProduct> orderedProductCollection;
```

Ordered_product

```
-- -----------------------------------------------------
-- Table `affablebean`.`ordered_product`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `affablebean`.`ordered_product` ;

CREATE  TABLE IF NOT EXISTS `affablebean`.`ordered_product` (
  `customer_order_id` INT UNSIGNED NOT NULL ,
  `product_id` INT UNSIGNED NOT NULL ,
  `quantity` SMALLINT UNSIGNED NOT NULL DEFAULT 1 ,
  PRIMARY KEY (`customer_order_id`, `product_id`) ,
  INDEX `fk_ordered_product_customer_order` (`customer_order_id` ASC) ,
  INDEX `fk_ordered_product_product` (`product_id` ASC) ,
  CONSTRAINT `fk_ordered_product_customer_order`
    FOREIGN KEY (`customer_order_id` )
    REFERENCES `affablebean`.`customer_order` (`id` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_ordered_product_product`
    FOREIGN KEY (`product_id` )
    REFERENCES `affablebean`.`product` (`id` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB
COMMENT = 'matches products with customer orders and records their quantity';
```

OrderedProduct

- Association table between product and customer order
  - @EmbeddedId
    - Persists the composite PK field as the ID of the object
    - Can't use Id for a composite PK class

```
public class OrderedProduct implements Serializable {
    private static final long serialVersionUID = 1L;
    @EmbeddedId
    protected OrderedProductPK orderedProductPK;
    @Basic(optional = false)
    @Column(name = "quantity")
    private short quantity;
    @JoinColumn(name = "product_id", referencedColumnName = "id", insertable = fa
    @ManyToOne(optional = false)
    private Product product;
    @JoinColumn(name = "customer_order_id", referencedColumnName = "id", insertab
    @ManyToOne(optional = false)
    private CustomerOrder customerOrder;
```

Customer entity

- One customer mapped to many customer orders
- MappedBy: the field at the other end of the relationship which references this (customer) end
- So each customer object has a collection of the related customer order objects

```
private String ccNumber;
@OneToMany(cascade = CascadeType.ALL, mappedBy = "customer")
private Collection<CustomerOrder> customerOrderCollection;
```

Category class

- Named Queries
  - Fixed queries
  - Mapped to specific class
  - Can include parameters

```
@NamedQueries({
    @NamedQuery(name = "Category.findAll", query = "SELECT c FROM Category c"),
    @NamedQuery(name = "Category.findById", query = "SELECT c FROM Category c WHERE c.id = :id"),
    @NamedQuery(name = "Category.findByName", query = "SELECT c FROM Category c WHERE c.name =
:name")})
```

  - See servlets:
    - getCategories
    - getProductsByCategoryName

Use of named query

- In GetCategories servlet

```
List<Category> categories = new ArrayList<>();
Query q = em.createNamedQuery("Category.findAll");
categories = q.getResultList();
```

- In GetProductsByCategoryName servlet

```
String category = request.getParameter("category");
Query q = em.createNamedQuery("Category.findByName");
q.setParameter("name", category);
Category c = (Category) q.getSingleResult();
```

- private static final long serialVersionUID = 1L;
  - The serialVersionUID is a universal version identifier for a Serializable class. Deserialization uses this number to ensure that a loaded class corresponds exactly to a serialized object. If no match is found, then an InvalidClassException is thrown.

Many-to-One

- OrderedProduct has a many-to-one relationship with Product and with CustomerOrder

```
@JoinColumn(name = "product_id", referencedColumnName = "id", insertable = false, updatable = false)
@ManyToOne(optional = false)
private Product product;
```

- 'product_id' is the foreign key of Product in the OrderedProduct table
- 'id' is the referenced column name in the related table (Product)

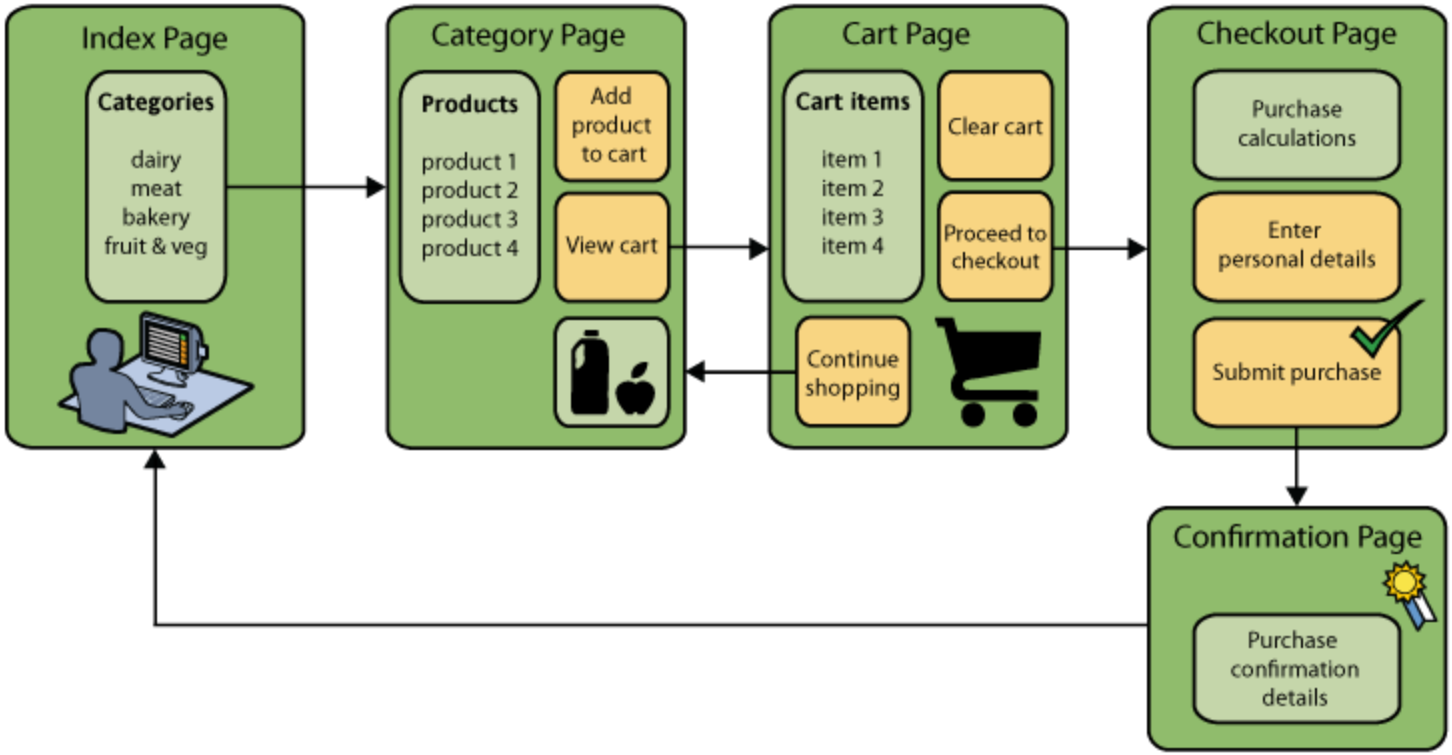
One-to-Many

- Use the attribute mappedBy so that the persistence engine knows how to join OrderedProduct and Product
- The mappedBy attribute exists on the other side of the relationship, which is the Product.
- In this example, the mappedBy attribute shows that an OrderedProduct instance's 'product' property maps to the Product instance.
- This means that the Products primary key will be a foreign key in the OrderedProduct table.
- In the actual Product object, though, we also have a Collection of OrderedProduct objects that it is linked to

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "product")
private Collection<OrderedProduct> orderedProductCollection;
```

AffableBean: e-commerce tutorial

- Based on the NetBeans e-commerce tutorial
- Basic flow is

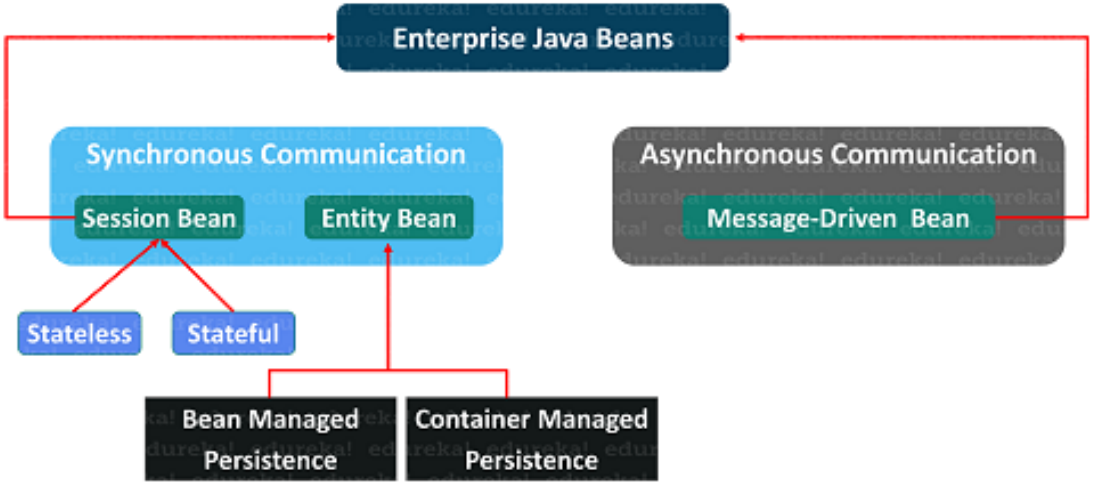


resources

- AffableBean.zip
  - NetBeans project
  - Contains a setup directory with database set files

Enterprise Beans

- The components that carry out the business logic / processes
- Use interfaces to make that functionality available to clients (internal and external to the application)
- They are managed by a container
- Three kinds:
  - Session Beans
  - Entity Beans
  - Message-driven Beans



Session Beans

- Business logic that can be invoked by a client
- Lifecycle / state managed by EJB container, session beans can be:
  - Stateless
    - Shared by multiple clients
    - The EJB container can pool them to provide to clients
  - Stateful
    - Client-specific instances
  - Singleton
    - One instance per application / shared across all clients

Examples of session beans

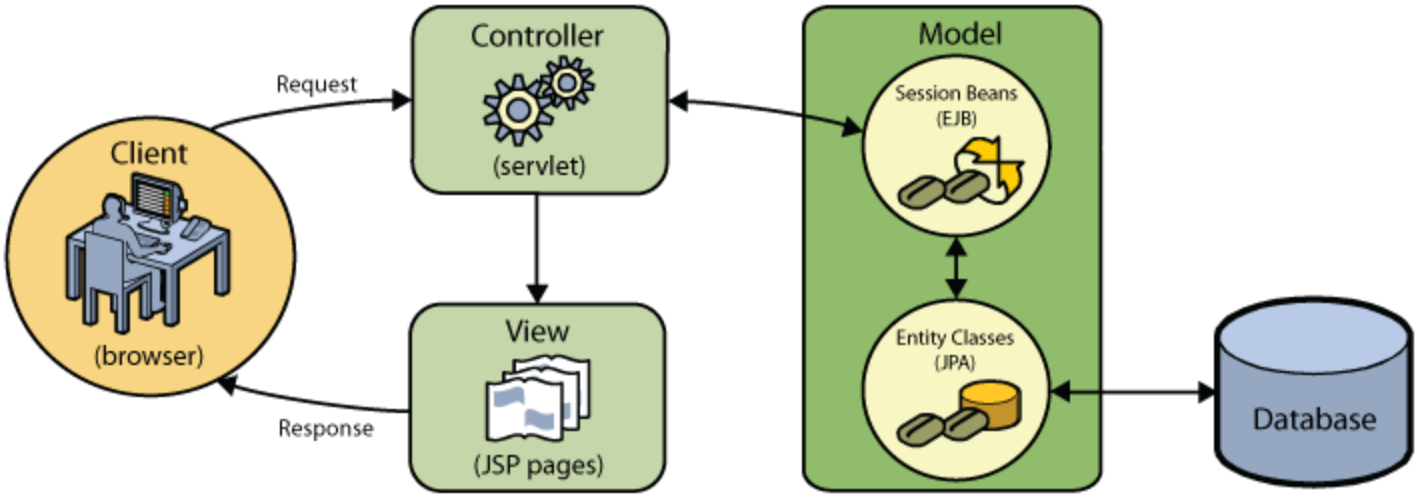
- A session bean in a human resources application that creates a new employee and assigns the employee to a particular department
- A session bean in an expense reporting application that creates a new expense report
- A session bean in an order entry application that creates a new order for a particular customer
- A session bean that manages the contents of a shopping cart in an e-commerce application (would be client-specific, so should be @stateful)

Message-driven Beans

- Allows clients to process messages asynchronously
- Usually a JMS listener – receives JMS messages
  - Mapped to a JMS queue
- Event-driven – when message arrives, the container calls the beans *onMessage* method to process the message

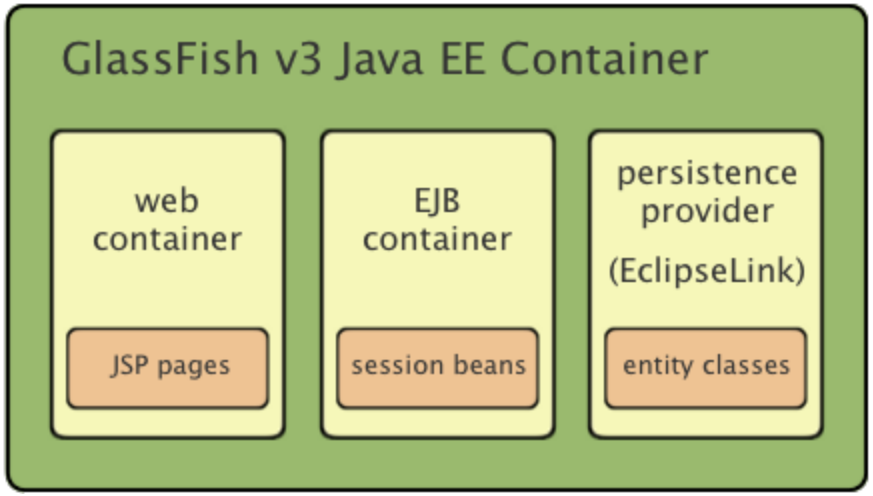
MVC architecture using session beans

- We need to create the entity beans, and also session beans, which will act as a *façade*, hiding the details of using JPA to manipulated (CRUD) the entity beans
- Session beans can also contain code (business logic) for the application



Java EE containers

- The entity classes are managed by the persistence provider
- The session beans are managed by the EJB container
- Views are rendered in JSP pages, which are managed by the web container.

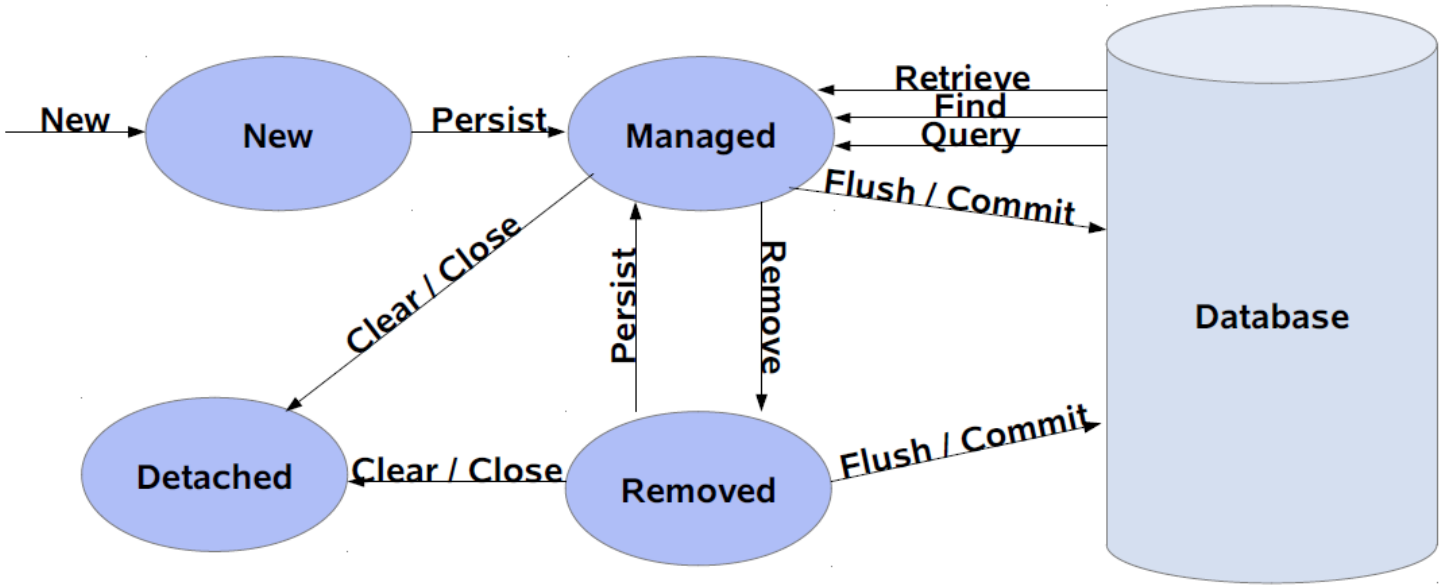


Entity Manager

- Entities are managed by an EntityManager
  - An instance of javax.persistence.EntityManager
- Each EntityManager instance is associated with a persistence context
  - The persistence context is a set of managed entity instances that exist in a particular data store
  - A persistence context defines the scope under which particular entity instances are created, persisted, and removed
- The EntityManager interface defines the methods that are used to interact with the persistence context

Entity life cycle

- Each arrow is an entity manager method (except new)
- Each oval shape is an entity state



A detached entity (a.k.a. a detached object) is **an object that has the same ID as an entity in the persistence store but that is no longer part of a persistence context**, e.g. because the EM which retrieved it was closed, or the object was received from outside the application

Container-managed Entity Manager

- A Java EE container manages the lifecycle of container-managed entity managers
- An EntityManager instance's persistence context is automatically propagated by the container to all application components that use the EntityManager instance within a single JTA transaction
- To obtain an EntityManager instance, inject the entity manager into the application component with the *javax.persistence.PersistenceContext* annotation
- ...
- @PersistenceContext
- EntityManager em;

Finding entities using entity manager

- The find method is used to look up entities in the data store by the entity's primary key

```
@PersistenceContext
EntityManager em;
public void enterOrder(int custID, Order newOrder)
{
        Customer cust = em.find(Customer.class, custID);
        cust.getOrders().add(newOrder);
        newOrder.setCustomer(cust);
}
```

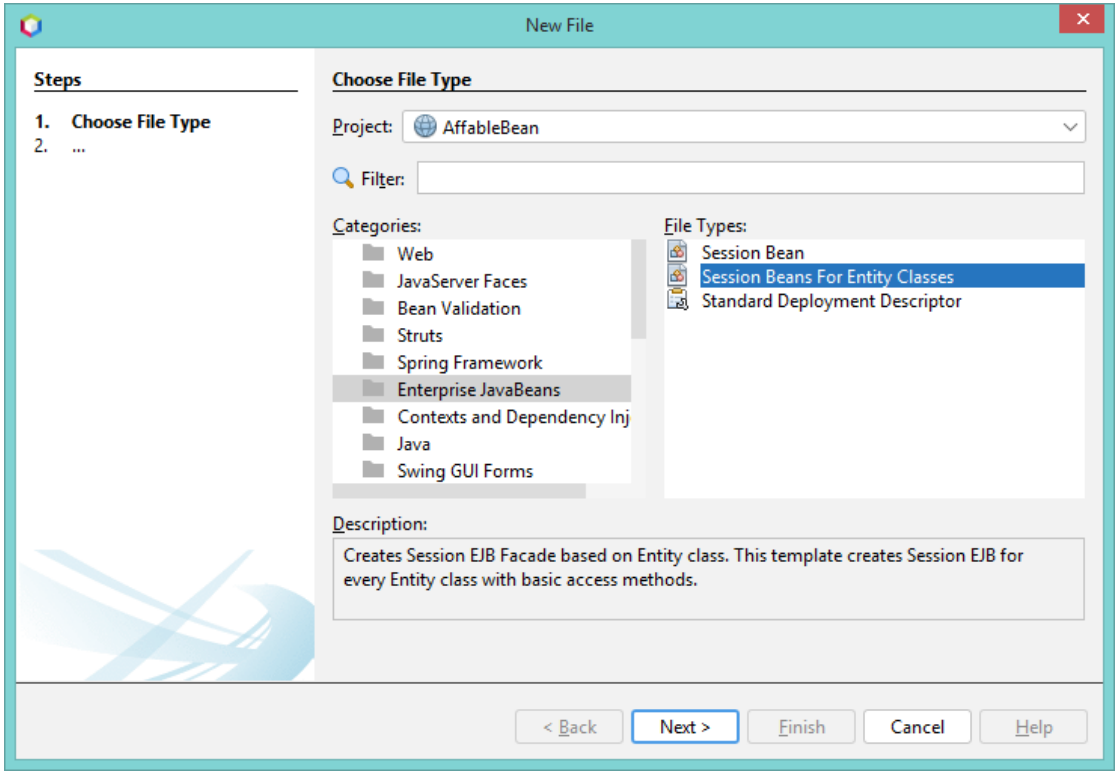
Application-managed Entity Manager

- **lifecycle of EntityManager instances is managed by the application**
- Applications create EntityManager instances here by using the createEntityManager method of javax.persistence.EntityManagerFactory
- To obtain an EntityManager instance, you first must obtain an EntityManagerFactory instance by injecting it into the application component by means of the javax.persistence.PersistenceUnit annotation, e.g.

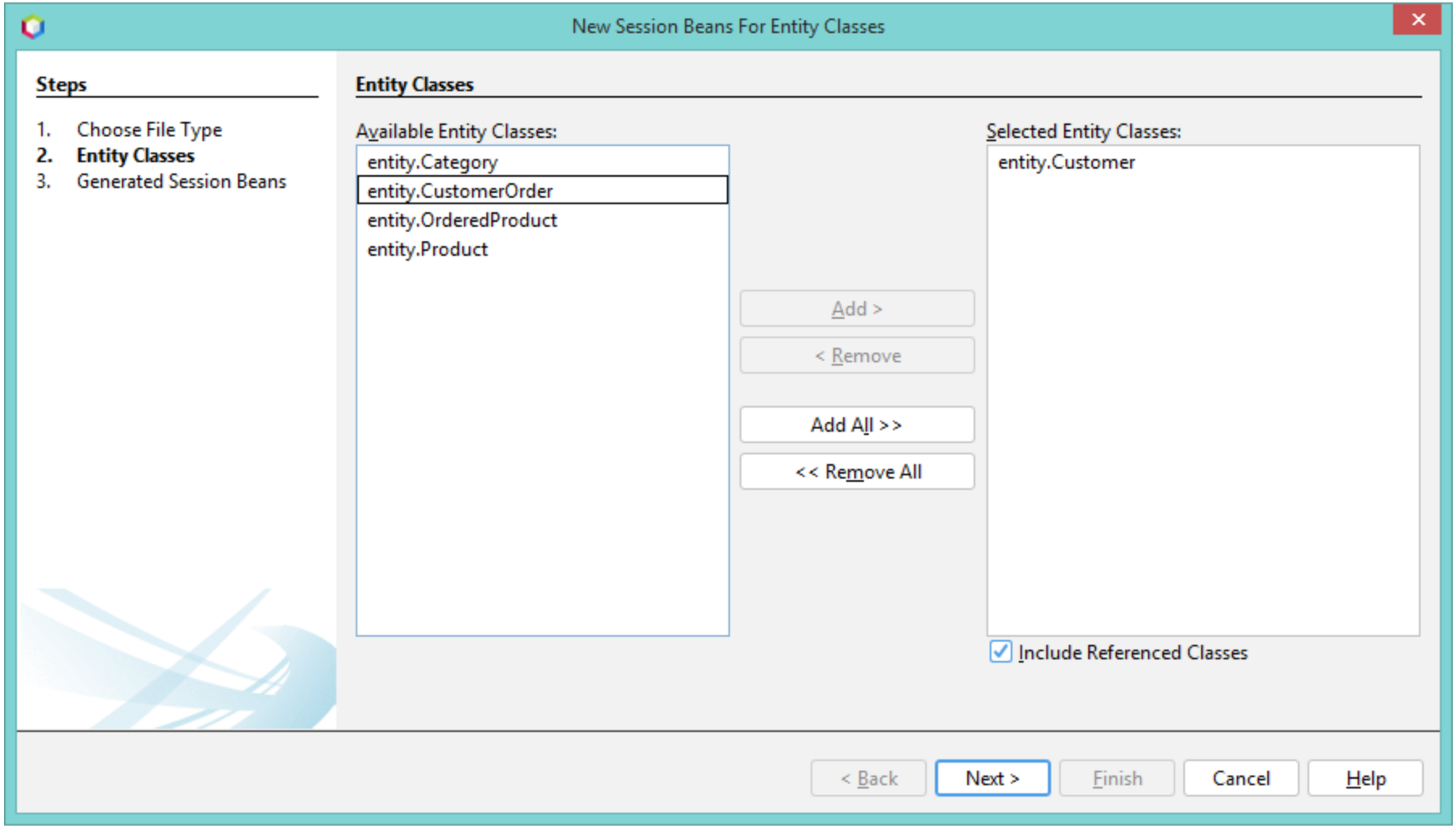
```
@PersistenceUnit (unitName = “myPU”)
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
   utx.begin();
   em.persist(SomeEntity);
   em.merge(AnotherEntity);
   em.remove(ThirdEntity);
   utx.commit();
} catch (Exception e) {
   utx.rollback();
}
```

Adding Session Beans

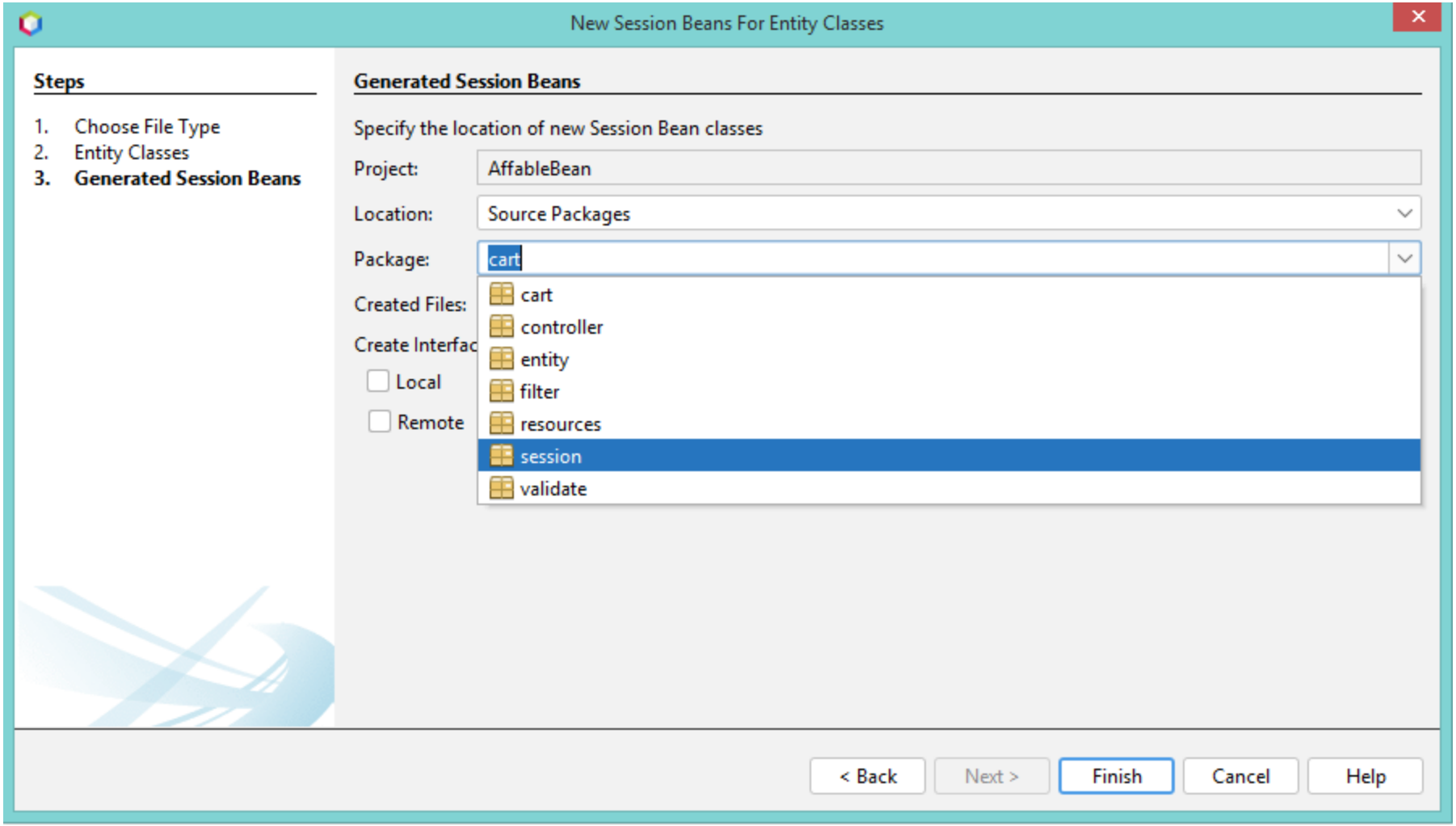
- In this app, session beans are used as a façade
  - Avoids tight coupling between clients and business objects
  - Cuts down on calls between client and server
  - Abstracts underlying (JPA primarily) interactions
  - Uses the container to manage the (session bean) life cycle



Choose entity classes



Select location for session beans



NetBeans generates session beans

- Code common to all classes is factored out into AnstractFacade
- The AbstractFacade class provides a few basic JPA methods to find entities of the related entity class
  - These methods use CriteriaQuery API

JPQL query

- For example a dynamic query such as

```
Query query = manager.createQuery("SELECT c FROM Car c WHERE c.color = :hexColor");
query.setParameter(“hexColor”, “FF0000”);
query.setMaxResults(100);
return query.getResultList();
```

Native SQL query

```
List<Car> cars = (List<Car>)em.createNativeQuery ("SELECT * FROM cars_table", Car.class)
.getResultList();


out.println("Cars: <br/><ul>");
for (Car car : cars)
 {
            out.println("</li>"+ car.getName()+": "+car.getMileage() +"</li>");
}
out.println("</ul>");
```

JPQL vs Criteria queries

- JPQL
  - Readable, concise, SQL-like query strings
  - Can use named queries defined using annotations in entity classes
- Criteria queries
  - Can be defined in business tier (session beans)
  - Used as a Java API, typesafe
  - Less readable
  - More complex to write, fast to run

CriteriaQuery

- Steps involved:
  - Create a CriteriaBuilder object
    - You have to use an EntityManager instance
  - Create a query object by creating an instance of the CriteriaQuery interface
    - This query object's attributes will be modified with the details of the query
  - Set the query root
    - Call the from method on the CriteriaQuery object
  - Specify type of the query result
    - Calling the select method of the CriteriaQuery object
  - Prepare the query for execution
    - Creating a TypedQuery<T> instance, specifying the type of the query result
  - Execute the query
    - Call the getResultList method on the TypedQuery<T> object

Example

```
List<Employee> employees = new ArrayList<>();

// use em to create CriteriaBuilder
CriteriaBuilder cb = em.getCriteriaBuilder();

// create a Query object
CriteriaQuery cq = cb.createQuery();

// set the query root - like the FROM part of a regular query
Root emp = cq.from(Employee.class);

// Specify what the type of the query result will be
cq.select(emp);

// Prepare the query for execution
Query q = em.createQuery(cq);

// Execute the query
employees = q.getResultList();
```

Example – find all

```
// Query for a List of data elements.
CriteriaQuery cq = cb.createQuery();
Root e = cq.from(Employee.class);
cq.select(e);
Query query = em.createQuery(cq);
List<Employee> result = query.getResultList();
```

Query for a List of element arrays

```
CriteriaQuery cq = cb.createQuery();
Root e = cq.from(Employee.class);
cq.multiselect(e.get("firstName"),employee.get("lastName"
));
Query query = em.createQuery(cq);
List<Object[]> result5 = query.getResultList();
```

Return user-defined objects

```
CriteriaQuery<EmployeeInfo> query =
criteriaBuilder.createQuery(EmployeeInfo.class);

Root<Employee> employee = query.from(Employee.class);
query.select(criteriaBuilder.construct(EmployeeInfo.class,
employee.get(Employee_.name), employee.get(Employee_.salary)));

List<EmployeeInfo> resultList =
entityManager.createQuery(query).getResultList();
```

https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/criteria-api-construct.html

User-defined class needed too

```
public class EmployeeInfo
{
  private String name;
  private double salary;

  public EmployeeInfo(String name, double salary)
{
    this.name = name; this.salary = salary;
}
 ..... }
```

- Further information (when needed):
  - https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/criteria-select.html
  - https://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/Criteria#Criteria_API

AffableBean walkthrough

- Points to look at:
  - Configuration using web.xml
    - Headers and footers
    - Context parameters
    - Session timeout
  - Session Beans using Façade pattern
  - Controller servlet with multiple URL patterns
    - Load on startup
    - init method
    - Use of view and userPath
  - Client-side validation
    - In checkout.jsp using jQuery validate function
  - Server-side validation
    - Using Validator class
- Language support
  - Throughout using <fmt:message
  - Resources defined in web.xml

Session Beans as Façade classes

- Uses the AbstractFacade class to implement common operations for Type<T> where <T> is a JPA entity
- Façade hides complexity of the (JPA) sub-system
  - Can use JPA as well if you want to
- Uses generics to provide common operations for all session bean classes extending AbstractFacade

```
public void create (T entity) {        getEntityManager().persist(entity);    }

   public void edit (T entity) {       getEntityManager().merge(entity);     }

   public void remove (T entity) {     getEntityManager().remove(getEntityManager().merge(entity));    }

   public T find (Object id) {      return getEntityManager().find(entityClass, id);    }

   public List<T> findAll () {       javax.persistence.criteria.CriteriaQuery cq =
                                              getEntityManager().getCriteriaBuilder().createQuery();
                                                    cq.select(cq.from(entityClass));
                                              return getEntityManager().createQuery(cq).getResultList(); }
```

Using the façade classes

- Look for example at the ControllerServlet
  - @EJB tells container to create an instance of the EJB (CategoryFacade)

```
@EJB
private CategoryFacade categoryFacade;
…
String categoryId = request.getQueryString();
…
Category selectedCategory = categoryFacade.find(Short.parseShort(categoryId));
```

- Implementation of the Façade design pattern, which is designed to hide complex call flows. Here it is largely used as a DAO (Data Access Object)

Controller Servlet

- Acts as the hub / router for the application
  - Maps to a number of URL patterns
- Note use of loadOnStartup
  - If this is positive the servlet is loaded when application is started, not on first time it is called
  - This allows the init() method to be used to retrieve the List of Category objects from the database and set as a session attribute
    - The product categories are therefore immediately available to index.jsp when the user opens the application on the browser
  - A number of servlets could implement loadOnStartup – they will be loaded in order of the integers provided 1,2,…

Application parameters

- In Java EE the ***web.xml*** file is used to store static parameters which are useful in the application
- Data which is common to the whole application (and doesn't change frequently if at all) is defined using <context-param>, while data confined to a particular servlet scope is defined using <init-param>
- web.xml is also used to store parameters / rules relating to security constraints, session configuration (e.g. timeout), headers and footers to be included in specified pages, pages to be displayed for particular error codes, how to authenticate users accessing secured resources,..

<context-param>

- A parameter bound to the application
- For example:

```
 <context-param>
        <description>The delivery surcharge applied to all
orders</description>
        <param-name>deliverySurcharge</param-name>
        <param-value>3.00</param-value>
    </context-param>
```

- Accessed in the init() method of ControllerServlet:

```
surcharge =
servletConfig.getServletContext().getInitParameter("deliverySurcharge");
```

<context-param>

- For example:

```
<context-param>
    <description>The relative path to category images</description>
    <param-name>categoryImagePath</param-name>
    <param-value>img/categories/</param-value>
 </context-param>
```

- Accessed in index.jsp:

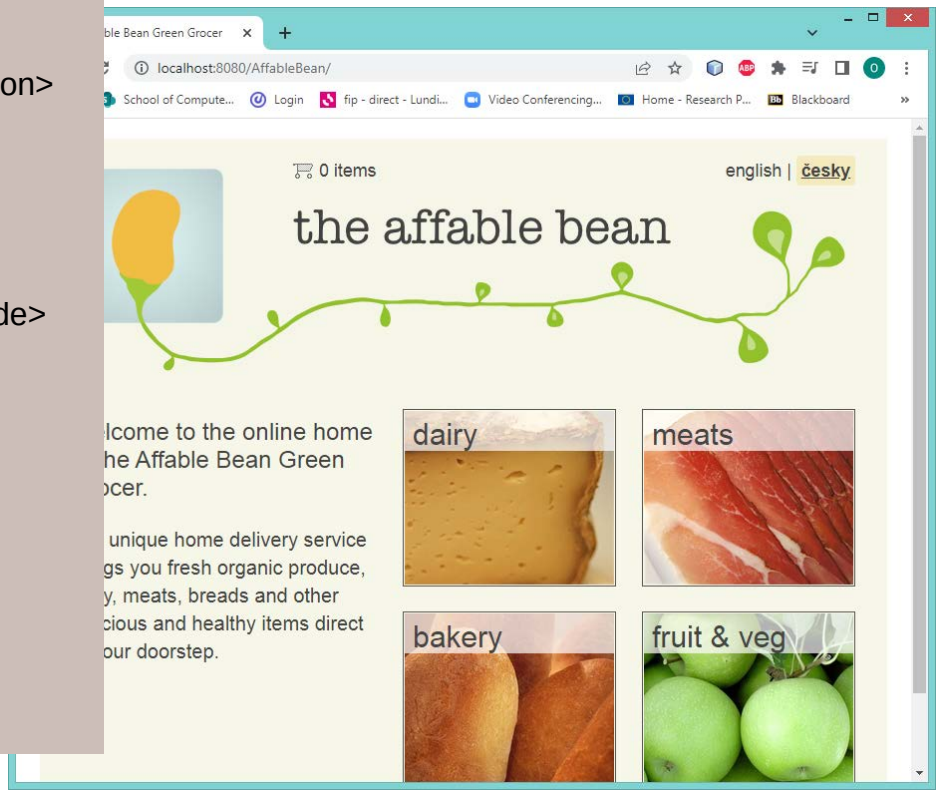
```
${initParam.categoryImagePath}
```

index.jsp

- See web.xml for configuration (header, footer)
  - Easy to apply same pattern to multiple pages

```
<jsp-config>
    <jsp-property-group>
      <description>JSP configuration for the store front</description>
      <url-pattern>/index.jsp</url-pattern>
      <url-pattern>/WEB-INF/view/*</url-pattern>
      <url-pattern>/WEB-INF/jspf/error/*</url-pattern>
      <include-prelude>/WEB-INF/jspf/header.jspf</include-prelude>
      <include-coda>/WEB-INF/jspf/footer.jspf</include-coda>
    </jsp-property-group>

    <jsp-property-group>
      …
    </jsp-property-group>
  </jsp-config>
```



index.jsp notes

- Note creation of a session attribute to help track calls from this page

```
<c:set var='view' value='/index' scope='session' />
```

- Use of

```
<fmt:message key='greeting' /></p>
```

  - Maps key to *localised* messages – performs replacement
- Location of localised message resources (files) defined in web.xml:

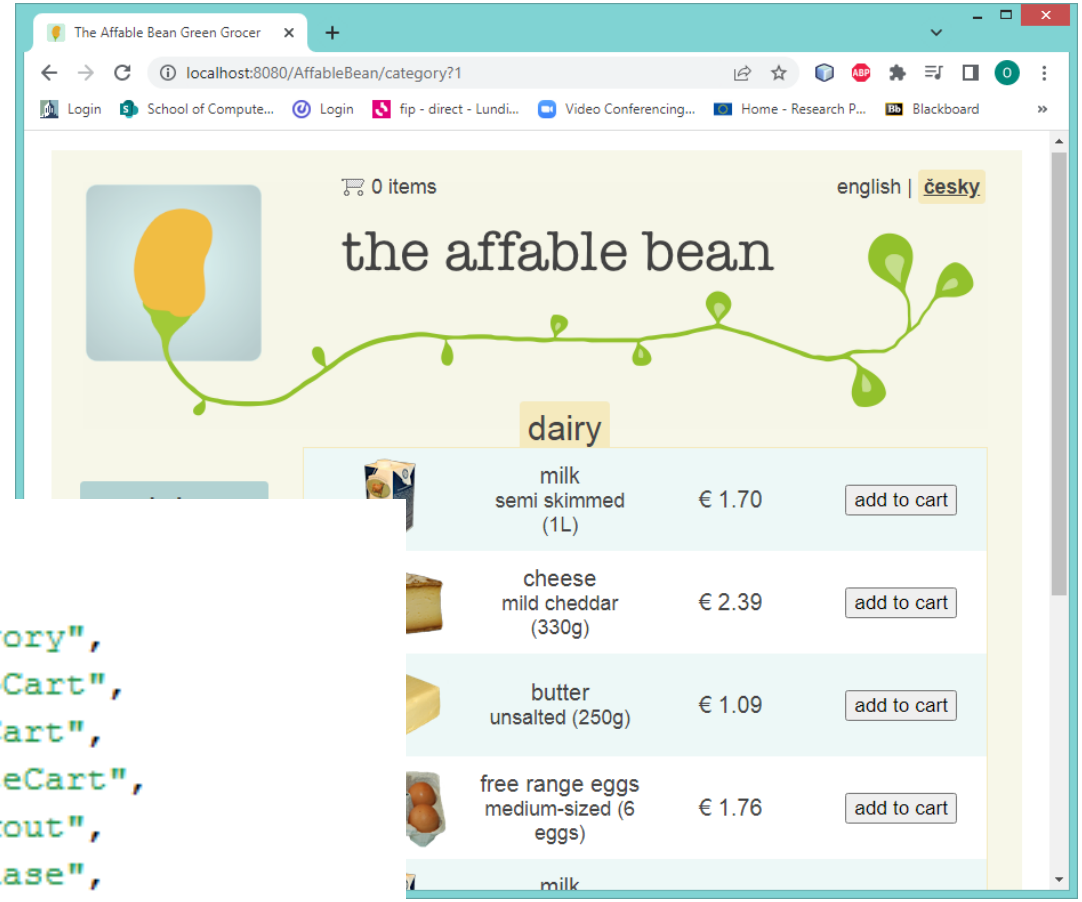
```
<context-param>
    <param-name>javax.servlet.jsp.jstl.fmt.localizationContext</param-name>
    <param-value>resources.messages</param-value>
</context-param>
```

- Files are in folder
  - src/main/resources/

Selecting a category

- Brings you to category page *category.jsp* in folder ../Web Pages/WEB-INF/view

- We get there via the *ControllerServlet*




```
@WebServlet(name = "Controller",
            loadOnStartup = 1,
            urlPatterns = {"/category",
                           "/addToCart",
                           "/viewCart",
                           "/updateCart",
                           "/checkout",
                           "/purchase",
                           "/chooseLanguage"})
public class ControllerServlet extends HttpServlet {
```

Handling /category URL in *ControllerServlet*

- uses the *request.getServletPath()* method to get the incoming (GET) request path
- *Request.getQueryString* returns the part after the path, so for example, here it would be just 1 if the full path was *http://localhost:8080/AffableBean/category?1*
- so it is OK to parse an int from this "*1*" string

```
Category selectedCategory;
Collection<Product> categoryProducts;


// if category page is requested
if (userPath.equals("/category")) {

    // get categoryId from request
    String categoryId = request.getQueryString();

    if (categoryId != null) {

        // get selected category
        selectedCategory = categoryFacade.find(Short.valueOf(categoryId));

        // place selected category in session scope
        session.setAttribute("selectedCategory", selectedCategory);

        // get all products for selected category
        categoryProducts = selectedCategory.getProductCollection();

        // place category products in session scope
        session.setAttribute("categoryProducts", categoryProducts);
    }
```

Forwarding from *ControllerServlet*

- *userPath* is "*category*" so this will brings us straight to *category.jsp*:

```
String url = "/WEB-INF/view" + userPath + ".jsp";

try {
    request.getRequestDispatcher(url).forward(request, response);
} catch (Exception ex) {
    ex.printStackTrace();
}
```

Managing sessions

- Timeout is defined in the web.xml file

```
<session-config>
    <session-timeout> 30
    </session-timeout>
</session-config>
```

- There is also a redirect in the SessionTimeoutFilter if the session is null (which it will be if the session has timed out)
- Note that we don't create a new session here if it doesn't exist:

```
HttpSession session = req.getSession(false);
```