Application architectures for data integration

Databases 2

Summary

- Goals: understanding database development in the context of architecture and application development
- Two-tier architectures (client-server)
- Three-tier architectures
  - Web: Web server & script engine
- Data integration in three tier architectures
- Requirements for a better integration approach

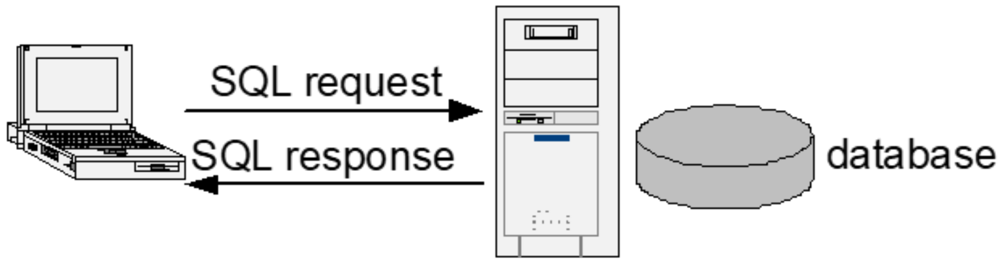
Architecture: definition and classification

- Architecture = mix of HW, SW and network resources
- Classification parameters
    - Type and functionality of HW resources
    - Topology of links connecting HW resources
    - Structure of software modules and relationship between each other

Brief history of distributed architectures

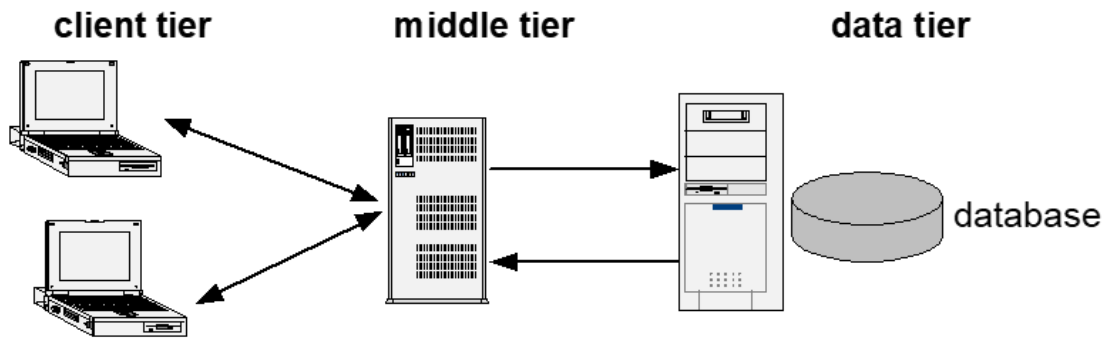
- 60’s-70’s: One-tier architectures
- 70’s-80’s: Client-Server architectures
- 80’s: Distributed architectures based on Remote Procedure Call (RPC)
- 90’s: OO distributed architectures
- Late 90’s: Internet and Web-based architectures
- Today: Distributed Web-based and mobile architectures, service oriented architectures (SOA, REST, Micro Services), Cloud and virtualized architectures, Application Service Provisioning (ASP)

Client server



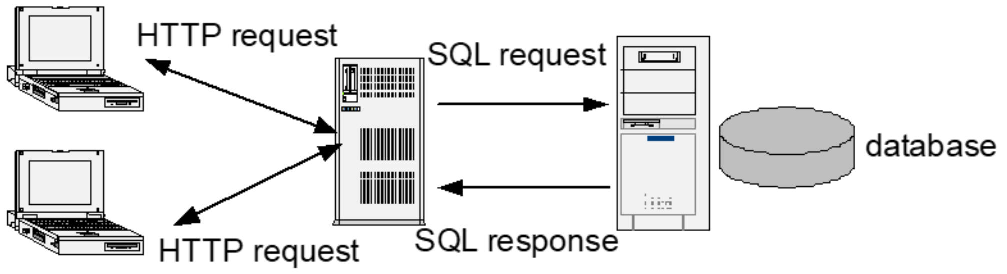
- Specialized HW
  - Server for data management
  - Client for presentation layout
- Topology: LAN with 1 or more servers and N clients
- Software structure: functional partitioning
  - Client SW sends requests to the server (database) by means of SQL queries. Client SW contains both business and presentation logic (e.g., pre-fetch, cache, client-side processing)
  - Server SW processes the query and responds sending the result set back to the client. Server SW only deals with data (e.g., integrity constraints, stored procedures)

Three-tier architectures



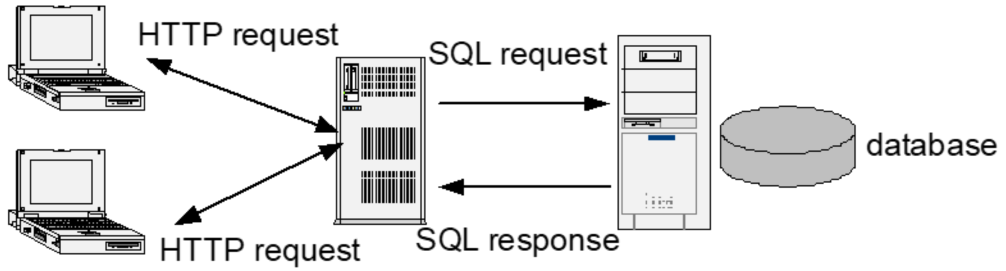
- New element → middle tier
- Adding the middle tier enables to achieve a better separation between the client and the server, since the middle tier:
  - Centralizes connections to the data server
  - Masks data model to the clients
  - Can be replicated to scale up
- This architecture has several variants, depending on the SW features of the middle tier (remote procedure call--RPC, object oriented, message oriented, Web)

Web “pure HTML” 3-tier architectures



- The client is a standard Web browser and it has to cope with the presentation layout only (**thin client**)
- The middle tier
  - includes a Web server that exploits a standard protocol (HTTP)
  - hosts the business logic for dynamically generating content from the raw data of the data tier
  - deals with the presentation layout (it can assemble presentation mark-up, i.e., interfaces)

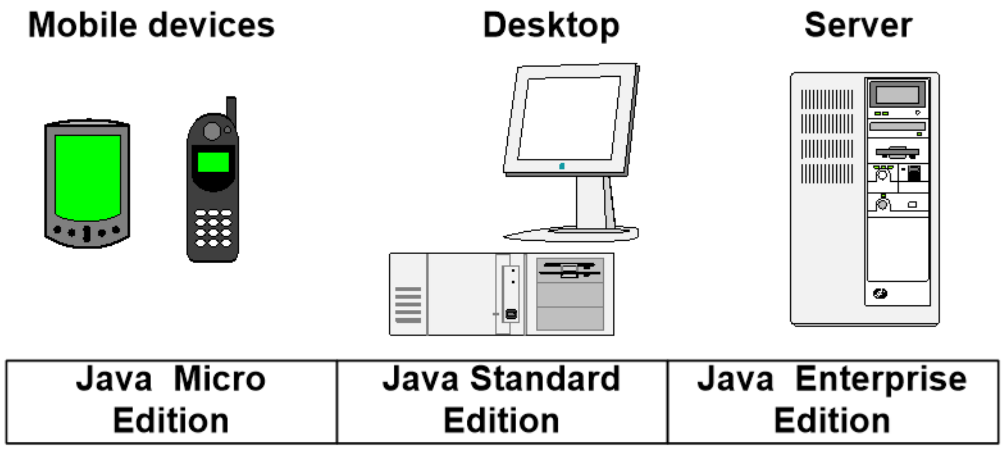
Rich Internet Applications



- The fusion of web and desktop applications
- Enabling technology: client side scripting (with JavaScript)
- **Fat client**: similar to client server but with standard communication protocol (HTTP, Web Socket), language (ECMAScript) and API (DOM, HTML5)
- Supported by the HTML 5 standard
- Features:
  - New interface event types, also specific to touch and mobile apps
  - Asynchronous interaction (AJAX)
  - Client-side persistent data
  - Disconnected / offline applications
  - Native multimedia and 3D support

Three-tier Web applications with Jakarta EE (formerly Java EE)

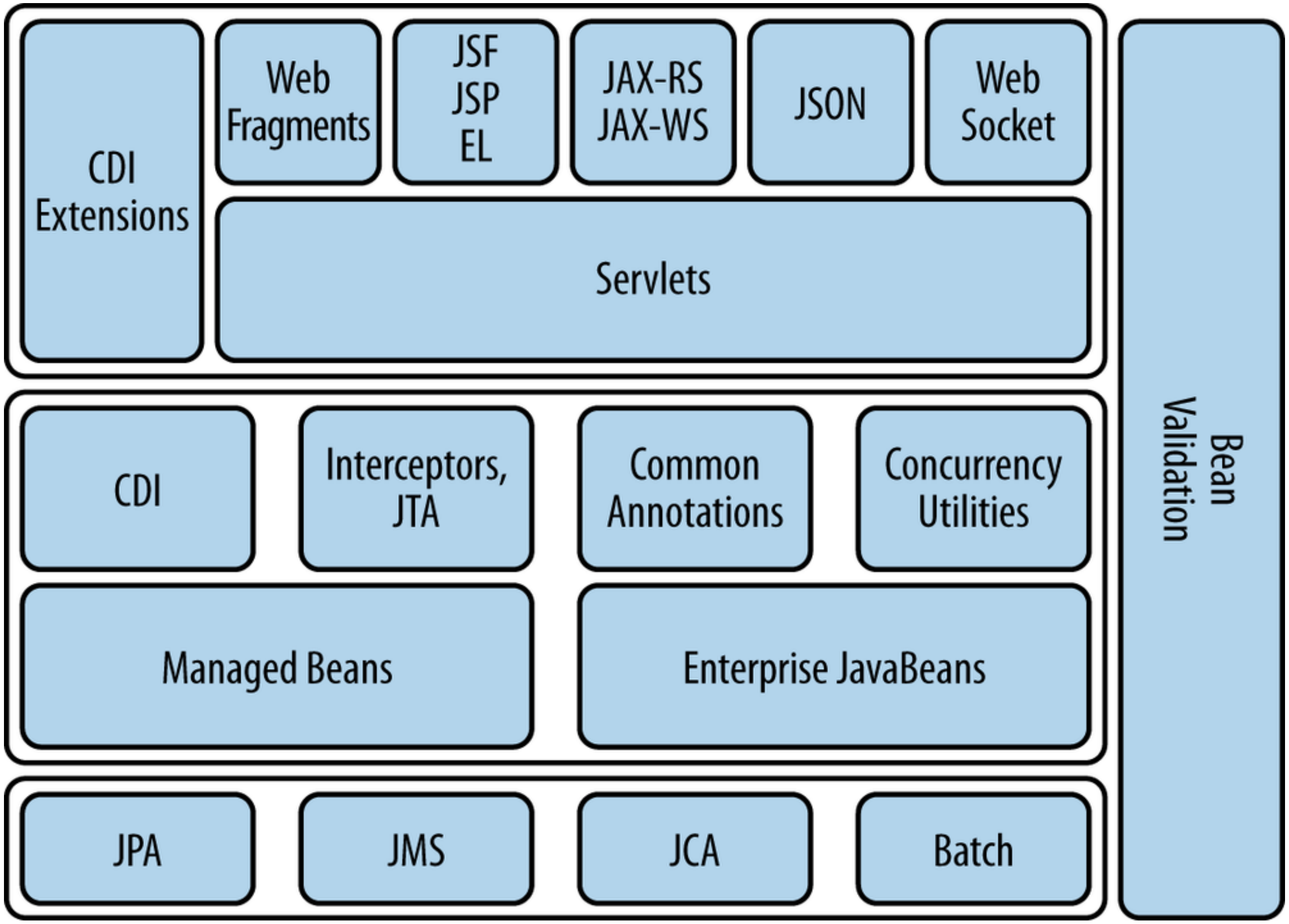
- JEE: platform aimed at the development, release and maintenance of three-tier Web applications
  - API and technology specifications
  - Development and release platforms
  - Reference software implementations
  - Compatibility Test Suite
  - Reference applications (blueprints)



Why JEE?

- Component based development
  - Developing enterprise applications by assembling components, i.e., small self-contained loosely coupled software modules
- Container services
  - Enhancing applications with many functional and non functional requirements provided off the shelf by the environment where components execute (security, transactionality, scalability, failure recovery, interoperability with external systems..)
- Declarative development
  - Declaring what is needed rather than programming it (declarative security, declarative transactions, declarative object to relation mapping)

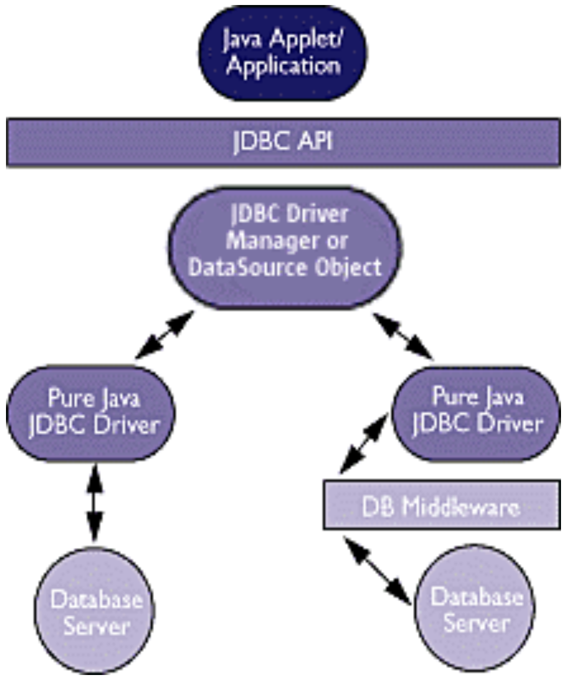
The JEE Stack



Java DataBase Connectivity (JDBC)

How people managed to do application on top of DB in the old days?

- The JDBC API was the first industry standard for database-independent connectivity between the Java programming language and databases
- The JDBC API makes it possible to do three things:
  - Establish a connection with a database or access any tabular data source
  - Send SQL statements
  - Process the results
- Still important but superseded by JTA/JPA



Servlet

- Servlets are the Java technology for Web application development (in the **presentation tier**)
    - offer a component-based, platform-independent method for building Web-based applications
    - have access to other Java APIs, including the JDBC API to access enterprise databases
    - are executed within a container (**servlet container**), which provides such services as concurrency and lifecycle management
- More on this in other courses, e.g., Web Technologies

Jakarta Enterprise Beans

- Jakarta Enterprise Beans (EJB, formerly: Enterprise Java Beans) technology is the server-side component architecture of JEE
- Focus on **Business Tier** component development
- Enables development of distributed, transactional, secure and portable applications based on Java technology
- EJB components execute within a container (**EJB container**)
- The EJB container offers services for lifecycle management, transaction management, replication and scaling
- Uses advanced language features
  - Annotations
  - Dependency injection
- EJB components can be used by the Web front end to interact with the business functions and data access services

Java Persistence API (JPA)

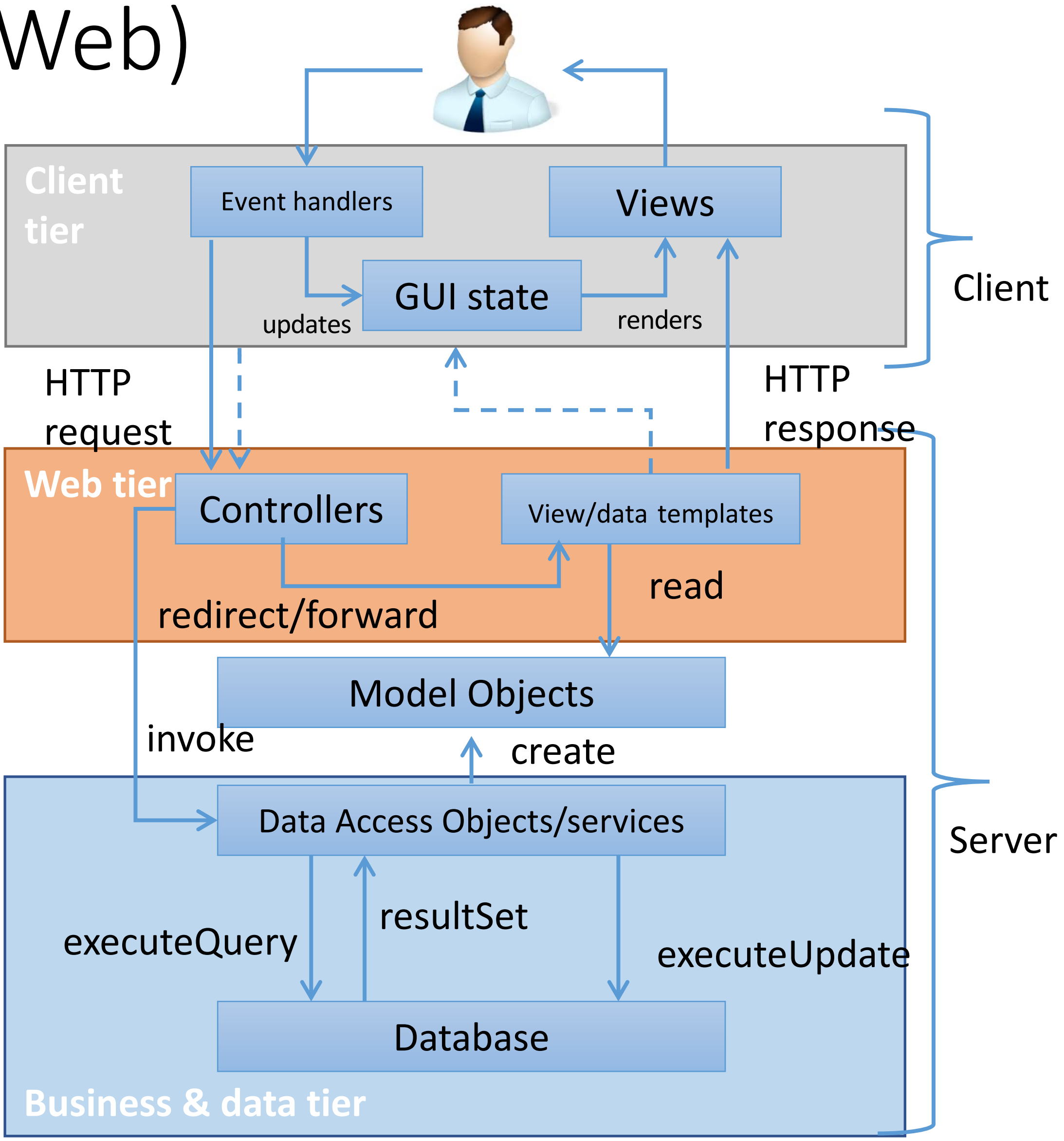
- The specification of an interface for mapping relational data to object oriented data in Java
- Integrated in both Java Standard Edition and Java Enterprise Edition
- Comprises:
  - The API implementation package javax.persistence
  - The Java compatible query language Java Persistence Query Language (JPQL)
  - The specification of the metadata for defining object relational mappings (ORMs)
- Supersedes the use of JDBC

Java Transaction API (JTA)

- An API for managing transactions **in Java**
- It allows a component to start, commit and rollback transactions in a resource-agnostic way
- With JTA, Java components can manage multiple resources (i.e. databases, messaging services) in a single transaction with a unique interaction model
- Transactional properties can be expressed
    - declaratively (with `@transactional` code annotation), method by method, eliminating the need to explicit transaction demarcation code
    - Programmatically, with the functions of the `UserTransaction` interface

How to connect an application to the database (3-tier, Web)

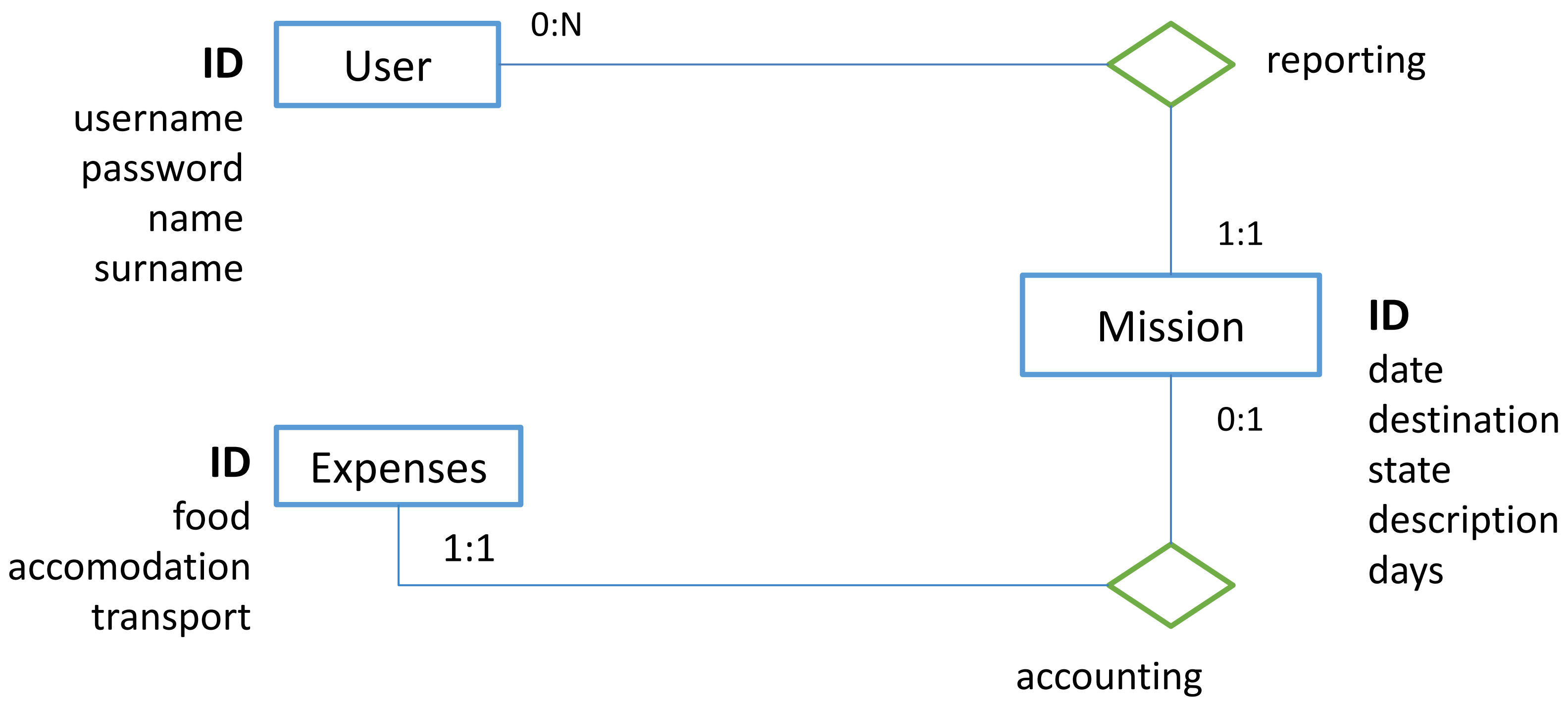
- **Client tier**: handles user's events, invokes the web tier and renders results in the interface (aka view)
- **Web tier**: dispatches (controllers) client calls to business logic components, formats and sends back results (extracted from model objects)
- **Business and data tier**: implements business logic and performs data queries and updates within transactions



Case study: expense report

A Web application supports the management of travel expenses. After logging in, the user accesses a HOME page where there is a list of travel missions; a mission belongs to a user and has a date, a place, a description, a number of days of duration, and a status ("open", “reported", "closed"). The list shows the date and place of the missions, which are sorted by date descending. On the HOME page there is a form, with which the user can create a new mission, by entering all the data, which are mandatory. A new mission is always in the “open” state. After creating a mission, the user is returned to the HOME page. When the user selects a mission in the list, a MISSION DETAIL page appears, showing all the mission data. If the mission is in the "open" state, a form appears for entering the expenses incurred during the mission; the form contains three fields: food costs, accommodation costs, transport costs. Sending the form data causes the mission status to change from "open“ to “reported", and the return to the MISSION DETAIL page. Total expense should be less than 100€ otherwise the report is rejected with an error. If the mission is in the “reported“ status, a CLOSE button appears and the user can click on it to declare that he has received the reimbursement; this causes the mission status to change from “reported" to "closed" and the redisplay of the MISSION DETAIL page. If the mission is in the "closed" status, the MISSION DETAIL page shows the mission data completed with the value of the three types of expenditure.

Database design



Logical database schema

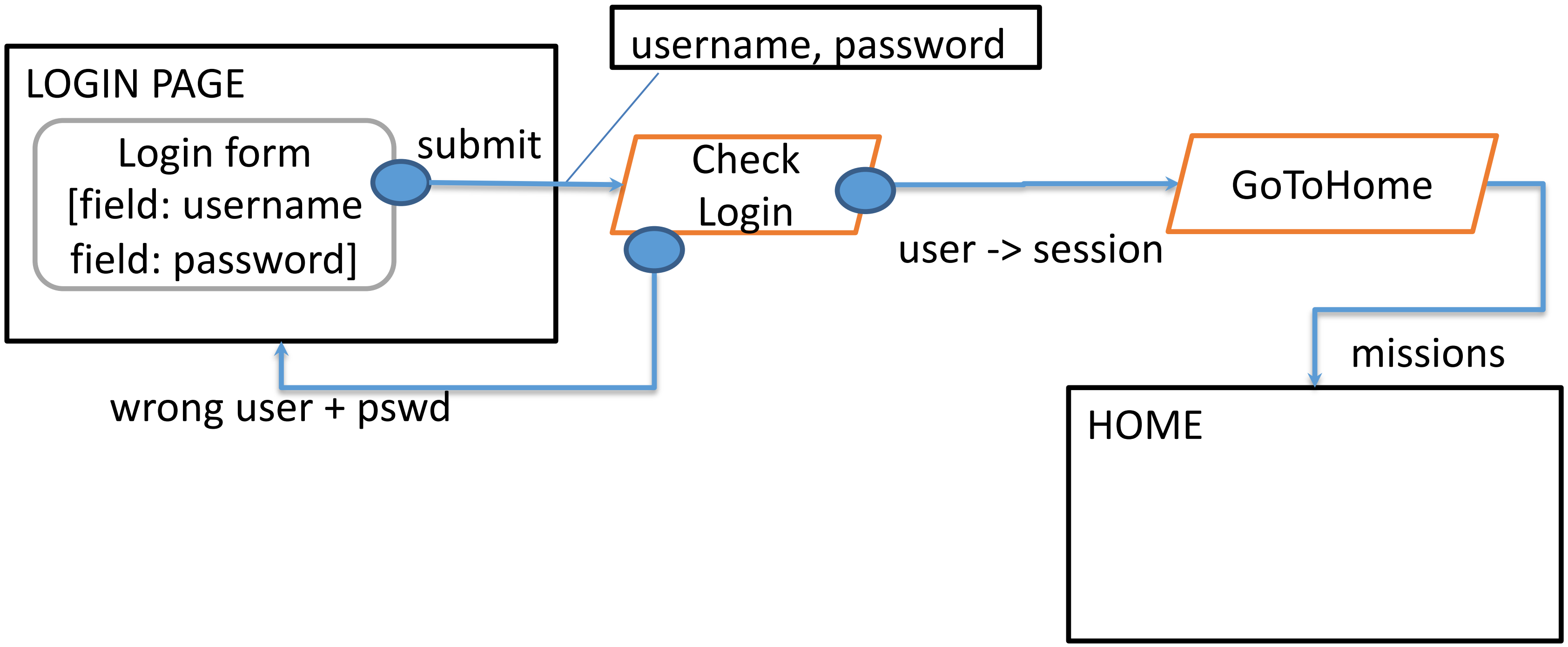
```
CREATE TABLE `user` (
`id` int(11) NOT NULL
AUTO_INCREMENT,
`username` varchar(45) NOT NULL,
`password` varchar(45) NOT NULL,
`name` varchar(45) NOT NULL,
`surname` varchar(45) NOT NULL,
PRIMARY KEY (`id`))
```

```
CREATE TABLE `mission` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`date` date NOT NULL,
`destination` varchar(45) NOT NULL,
`state` int(11) NOT NULL DEFAULT '0',
`description` varchar(45) NOT NULL,
`days` int(11) NOT NULL,
`reporter` int(11) NOT NULL,
PRIMARY KEY (`id`),
CONSTRAINT `id_reporter` FOREIGN KEY
(`reporter`) REFERENCES `user` (`id`)
ON DELETE CASCADE ON UPDATE CASCADE)
```

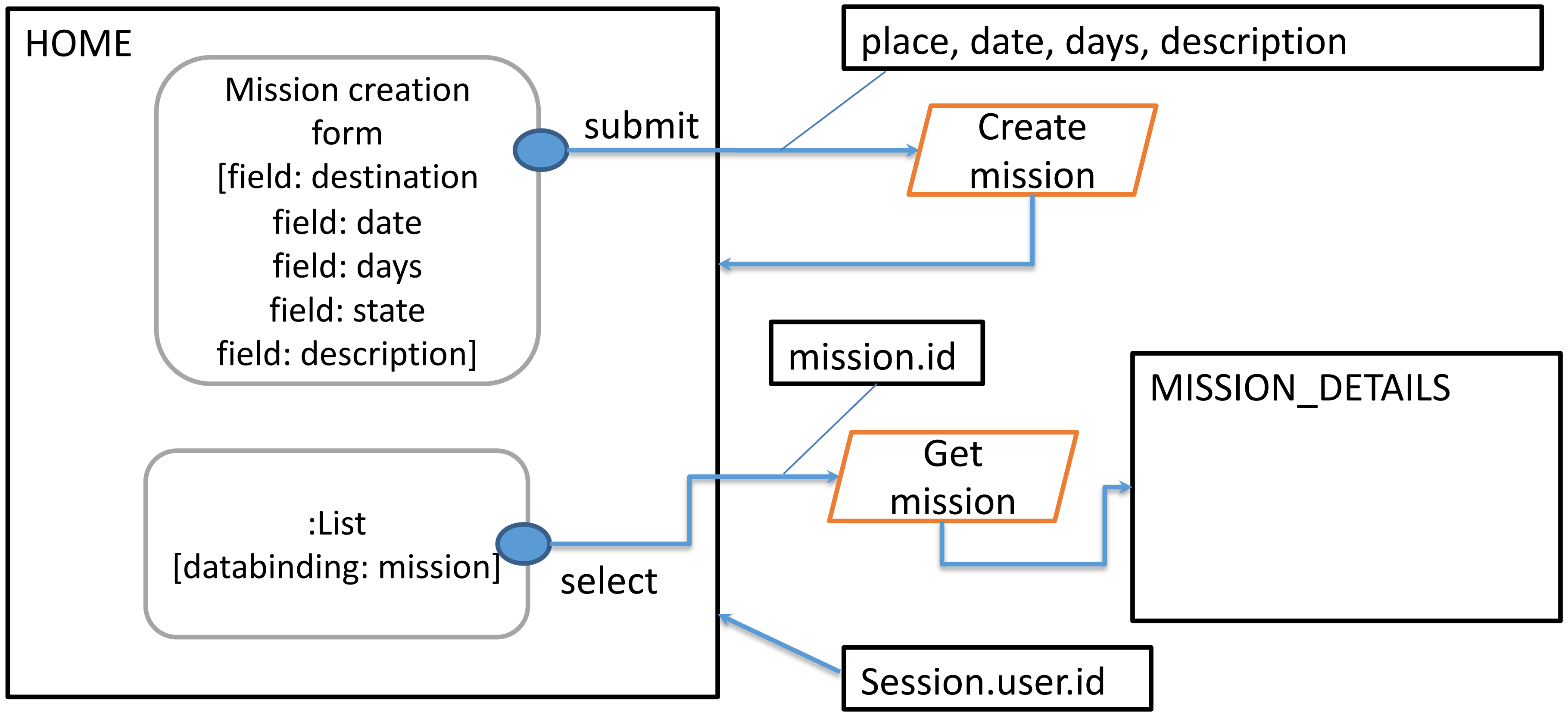
Logical database schema

```
CREATE TABLE `expenses`
(
`id` int(11) NOT NULL AUTO_INCREMENT,
`food` decimal(19,4) NOT NULL,
`accommodation` decimal(19,4) NOT NULL,
`transport` decimal(19,4) NOT NULL,
`mission` int(11) NOT NULL,
 PRIMARY KEY  (`id`),  KEY `id` (`mission`),
 CONSTRAINT `id` FOREIGN KEY (`mission`) REFERENCES
 `mission` (`id`) ON DELETE CASCADE ON UPDATE CASCADE
 CONSTRAINT `CHK_TotalExp` CHECK ((((`food` +
  `accomodation`) + `transport`) < 100.00))
)
```

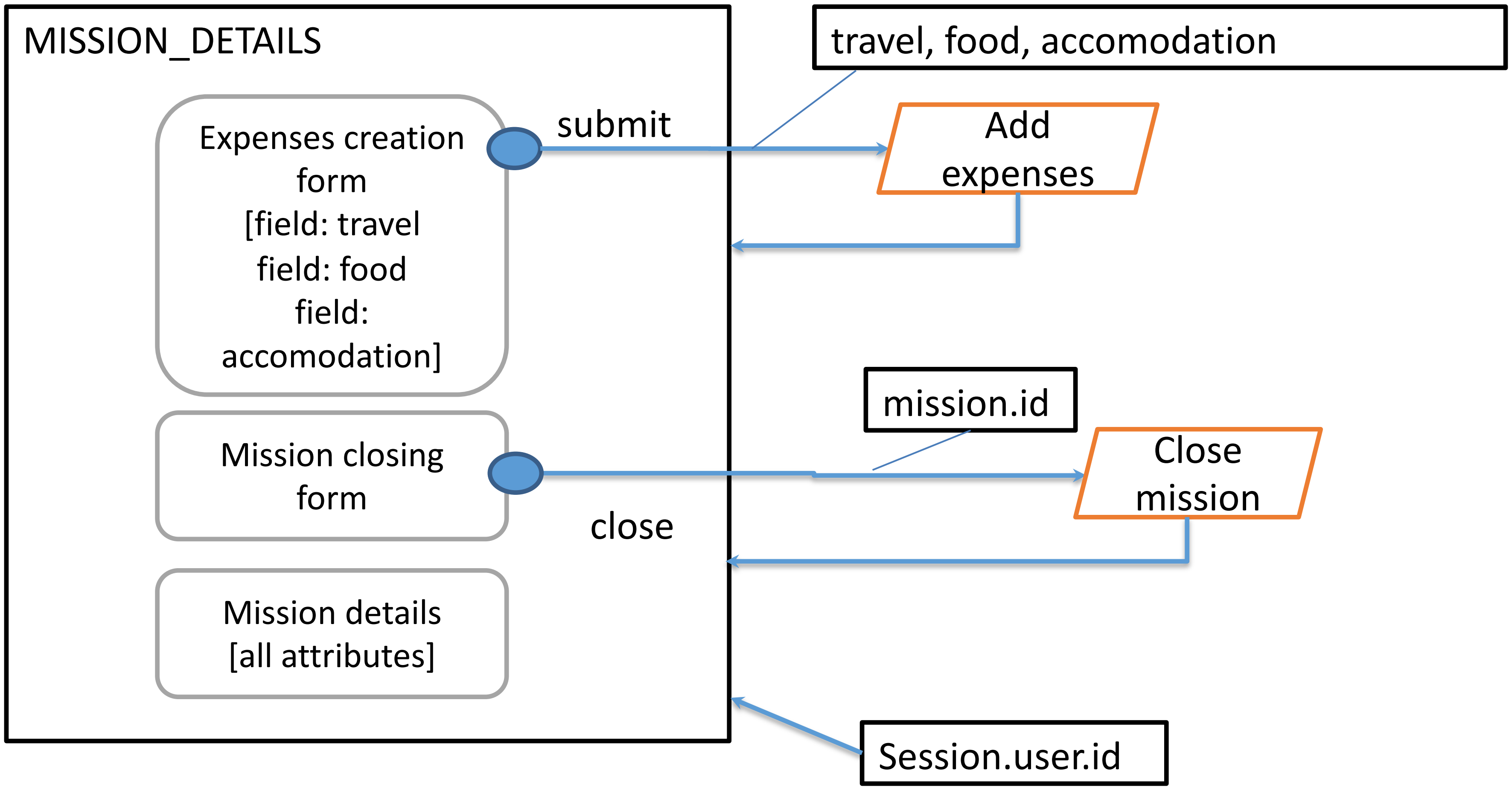
Application design



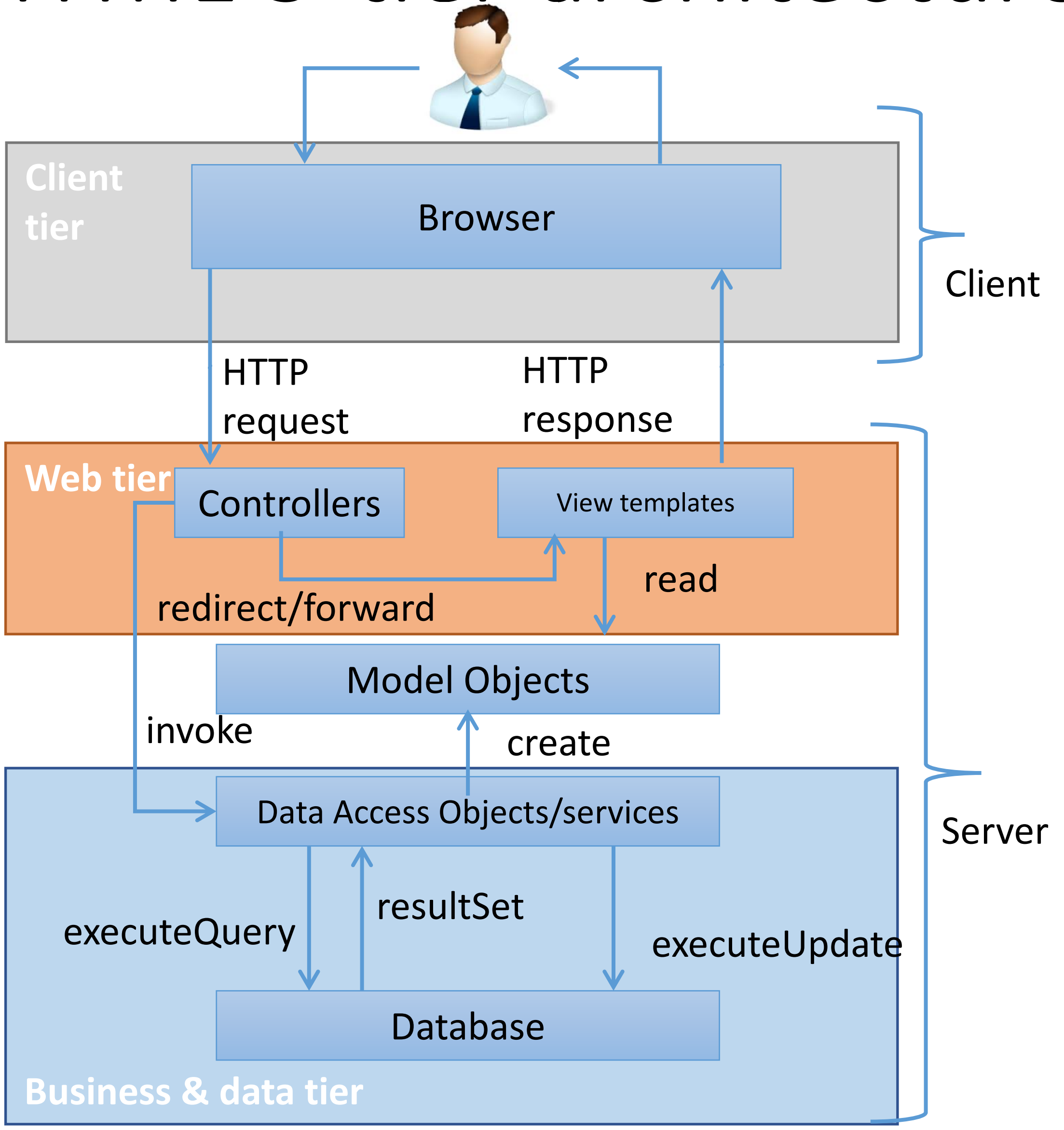
Application design


HOME
Mission creation form [field: destination field: date field: days field: state field: description]
submit
place, date, days, description
Create mission
:List [databinding: mission]
select
mission.id
Get mission
MISSION_DETAILS
Session.user.id

Application design


MISSION_DETAILS
Expenses creation form [field: travel field: food field: accomodation]
submit
Mission closing form
close
Mission details [all attributes]
travel, food, accomodation
Add expenses
mission.id
Close mission
Session.user.id

Pure HTML 3-tier architecture



Components

- Model objects (Beans)
  - User
  - Mission
  - ExpenseReport
- Data Access Objects (Classes)
  - UserDAO
    - checkCredentials(usrname, pwd)
  - MissionDAO
    - createMission(mission, userid)
    - findMissionsByUser(userid)
    - findMissionById(missionid)
    - changeMissionStatus(missionid, status)
  - ExpensesDAO
    - addExpenseReport(expenseReport, mission)
    - findExpensesForMission(missionId)
- Controllers (servlets)
  - CheckLogin
  - GoToHomePage
  - CreateMission
  - GetMissionDetails
  - CreateExpenses
  - CloseMission
  - Logout
- Views (Templates)
  - Login
  - Home
  - Mission Details

Database connection in web tier

In the good old days you had to get connected to the DB, after you will need to close it. And a lot of stuff had be done manually

- Web.xml configuration file in the WEB-INF folder of the web application server

```
<context-param>
  <param-name>dbUrl</param-name>
  <param-alue>
    jdbc:mysql://localhost:3306/db_mission
 </param-value>
</context-param>
 <context-param>
    <param-name>dbUser</param-name>
    <param-value>user</param-value>
</context-param>
<context-param>
    <param-name>dbPassword</param-name>
    <param-value>pass</param-value>
</context-param>
<context-param>
    <param-name>dbDriver</param-name>
    <param-value>
    com.mysql.cj.jdbc.Driver</param-value>
 </context-param>
```

- Utility functions called by all controllers that need be connected to the database

```
public class ConnectionHandler {

 public static Connection getConnection(ServletContext
     context)  throws UnavailableException {
 Connection connection = null;
 try {
  String driver = context.getInitParameter("dbDriver");
  String url = context.getInitParameter("dbUrl");
  String user = context.getInitParameter("dbUser");
  String password =
    context.getInitParameter("dbPassword");
  Class.forName(driver);
  connection = DriverManager.getConnection(url, user,
                                          password);
 } catch (ClassNotFoundException e) {
 throw new UnavailableException("Can't load driver");
 } catch (SQLException e) {
 throw new UnavailableException("Couldn't get db
                                connection");
 }
 return connection;
}

public static void closeConnection(Connection
connection) throws SQLException {
  if (connection != null) { connection.close();}
 }
}
```

(Repetitive) code in the controllers

- At initialization

```
public void init() throws ServletException {
  connection = ConnectionHandler.getConnection(
  getServletContext());
}
```

- At termination

```
public void destroy() {
try {
      ConnectionHandler.closeConnection(connection);
 } catch (SQLException e) {
   e.printStackTrace();
 }
}
```

- Every controller repeats the same code for acquiring and releasing the connection
- Connections are not handled efficiently (one per controller, stored as a data member of the controller class)
- Forgetting to release a connection is costly

Persistent data: extraction

```
// Method of the business object MissionDAO
public List<Mission> findMissionsByUser(int userId) throws SQLException {

 List<Mission> missions = new ArrayList<Mission>();
 String query = "SELECT * from mission where reporter = ? ORDER BY date DESC";

 try (PreparedStatement pstatement = connection.prepareStatement(query);) {
      pstatement.setInt(1, userId);
      try (ResultSet result = pstatement.executeQuery();) {
        while (result.next()) {
         Mission mission = new Mission();

        // Copy cursor data into Java bean objects...
         mission.setId(result.getInt("id"));
         mission.setStartDate(result.getDate("date"));
         mission.setDestination(result.getString("destination"));
         mission.setStatus(result.getInt("status"));
         mission.setDescription(result.getString("description"));
         mission.setDays(result.getInt("days"));
         mission.setReporterID(userId);
         missions.add(mission);
     }
   }
  } // exception handling omitted for brevity
 return missions;
}
```

query execution, which results a generic object of type ResultSet

Note: JDBC type is **ResultSet**

Application type is **List<Mission>**

Then we will need to "convert" it to the object we need

Persistent data: creation

```
// Method of the business object MissionDAO

public void createMission(Date startDate, int days, String destination,
                          String description, int reporterId)
 throws SQLException {

  String query = "INSERT into mission (date, destination, status,
           description, days, reporter) VALUES(?, ?, ?, ?, ?, ?)";
  try (PreparedStatement pstatement =
    connection.prepareStatement(query);) {

    // Copy Java parameters into SQL binding variables
    pstatement.setDate(1, new java.sql.Date(startDate.getTime()));
    pstatement.setString(2, destination);
    pstatement.setInt(3, MissionStatus.OPEN.getValue());
    pstatement.setString(4, description);
    pstatement.setInt(5, days);
    pstatement.setInt(6, reporterId);
    pstatement.executeUpdate();
   }
}
```

Persistent data: modification

```
// Method of the missionDAO business object
public void changeMissionStatus(int missionId,
        MissionStatus missionStatus) throws SQLException {

String query = "UPDATE mission SET status = ? WHERE id = ? ";
try (PreparedStatement pstatement =
       connection.prepareStatement(query);)
 {
       // copy the data to modify into the SQL query
       pstatement.setInt(1, missionStatus.getValue());
       pstatement.setInt(2, missionId);
       pstatement.executeUpdate();
 }
}
```

- Note the the `try` with resource clause to avoid the manual management of JDBC objects, e.g., statements and result sets
  - It will automatically close all the resources that implement the AutoCloseable interface

Transactions

```
// Method of expenseDAO business object
public void addExpenseReport(ExpenseReport expenseReport, Mission mission)
                          throws SQLException, BadMissionForExpReport {
 // Check that the mission exists and is in OPEN state
  ...
 MissionsDAO missionDAO = new MissionsDAO(connection);
 String query = "INSERT into expenses (food, accomodation, transport, mission)
                VALUES(?, ?, ?, ?)";
                                 //Delimit the transaction explicitly
 connection.setAutoCommit(false); //Override default commit after each statement
 try (PreparedStatement pstatement = connection.prepareStatement(query);) {
        pstatement.setDouble(1, expenseReport.getFood());
        pstatement.setDouble(2, expenseReport.getAccomodation());
        pstatement.setDouble(3, expenseReport.getTransportation());
        pstatement.setInt(4, expenseReport.getMissioId());
        pstatement.executeUpdate();    // 1st update
                                        // 2nd update, to be executed atomically
        missionDAO.changeMissionStatus(expenseReport.getMissioId(),
                                        MissionStatus.REPORTED);
        connection.commit();
    } catch (SQLException e) {
        connection.rollback(); // if update 1 OR 2 fails, roll back all work
        throw e;
      }
 } finally {connection.setAutoCommit(true);} // reset to standard
}
```

Wouldn’t it be better if

- A database query returned results already encoded in the right application type
- Creating an object created a tuple without the need of copying data manually
- Updating an object modified the matching tuple automatically
- Connections and data access objects were dispatched to the components that need them transparently to the application
- Methods calls could join transactions automatically based on the need

Problem-solution matrix

Requirement	Technology	How to
Avoiding the copy of data from query result cursor to native application types	JPA Object Relational Mapping	Java classes and relational tables can be associated via a declarative mapping
Avoiding the copy of data from program to UPDATE and INSERT SQL statement	JPA Object Relational Mapping	Creation and changes to Java objects can be persisted automatically
Avoiding the manual management of connections	JTA transactions EJB dependency injection	Connection object masked by higher level objects (EntityManager) automatically injected into the components that need them
Avoiding manual demarcation of transactions	JTA transactions EJB container managed transactions	Global transactions managed by the container and propagated to object methods that can “join” them

Data extraction with JPA

```
// method of missionService EJB
public List<Mission> findMissionsByUser(int userId) {
 Reporter reporter = em.find(Reporter.class, userId);
 List<Mission> missions = reporter.getMissions();
 return missions;
}
```

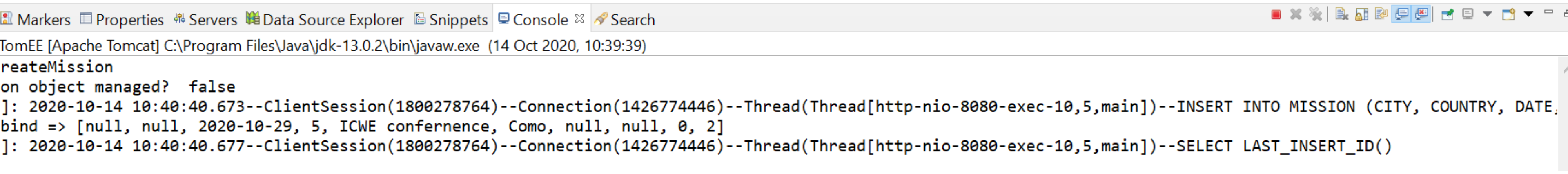
No manuel convertion, the right type of data is returned

Data creation with JPA

```
public void createMission(Date startDate, int days, String
destination, String description, int reporterId) {
 Reporter reporter = em.find(Reporter.class, reporterId);
 Mission mission = new Mission(startDate, days, destination,
                               description, reporter);
 reporter.addMission(mission);
 em.persist(reporter);
}
```

The “magic” of JPA

- Apparently no SQL at all, but the SQL is generated under the hood



```
TomEE [Apache Tomcat] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe  (14 Oct 2020, 10:39:39)
reateMission
on object managed?  false
]: 2020-10-14 10:40:40.673--ClientSession(1800278764)--Connection(1426774446)--Thread(Thread[http-nio-8080-exec-10,5,main])--INSERT INTO MISSION (CITY, COUNTRY, DATE,
bind => [null, null, 2020-10-29, 5, ICWE confernence, Como, null, null, 0, 2]
]: 2020-10-14 10:40:40.677--ClientSession(1800278764)--Connection(1426774446)--Thread(Thread[http-nio-8080-exec-10,5,main])--SELECT LAST_INSERT_ID()
```

Data modification with JPA

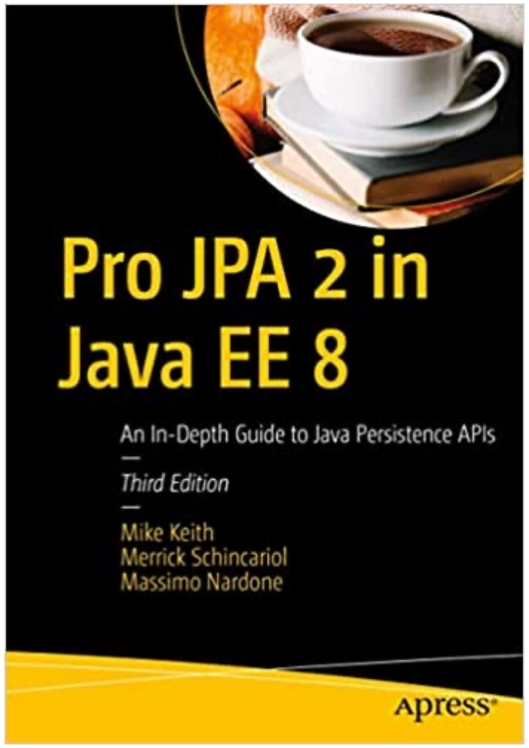
```
// Method of the missionService business object
public void reportMission(int missionId,
                          MissionStatus  missionStatus) {
  Mission mission = em.find(Mission.class, missionId);
  mission.setStatus(MissionStatus.REPORTED);
}
```

Transaction management in JPA

```
public void addExpenseReport(Expense expenseReport, int missionId)
throws BadMissionForExpReport {

// Check that the mission exists and is in OPEN state
  Mission mission = em.find(Mission.class, missionId); // now mission is managed

  if (mission == null | mission.getStatus() != MissionStatus.OPEN) {
  throw new BadMissionForExpReport("Mission cannot introduce expense report");
  }

  try { // this code is transactional!
   mission.setStatus(MissionStatus.REPORTED);
   mission.setExpense(expenseReport);
   em.persist(mission);
 } catch (PersistenceException e) {
   e.printStackTrace(); // used for debugging
 }
}
```

References

- The Java EE documentation, available at:
  - https://docs.oracle.com/javaee/7/index.html
- Bibliography textbook, Chapter 3 and Chapter 6


Pro JPA 2 in Java EE 8

An In-Depth Guide to Java Persistence APIs

Third Edition

Mike Keith
Merrick Schincariol
Massimo Nardone

Apress

Java Persistence API
Object Relational Mapping

Application data integration

How do I map java objects to data rows in a table

- The end-point of every web application is the DBMS
  - In many cases the database exists before and independently of the applications that use it
  - Moving data back and forth between a DBMS and the object model is **harder** than it needs to be
  - **A lot of repetitive code** is spent to convert row and column data into objects

Object Model vs. Relational Model

- The technique of bridging the gap between the object model and the relational model is known as **object-relational mapping** (ORM)
- ORM techniques try to map the concepts from one model onto another
  - **Impedance mismatch:** The challenge of mapping one model to the other lies in the concepts in one model for which there is no logical equivalent in the other
- A **mediator** is needed to manage the automatic transformation of one into the other

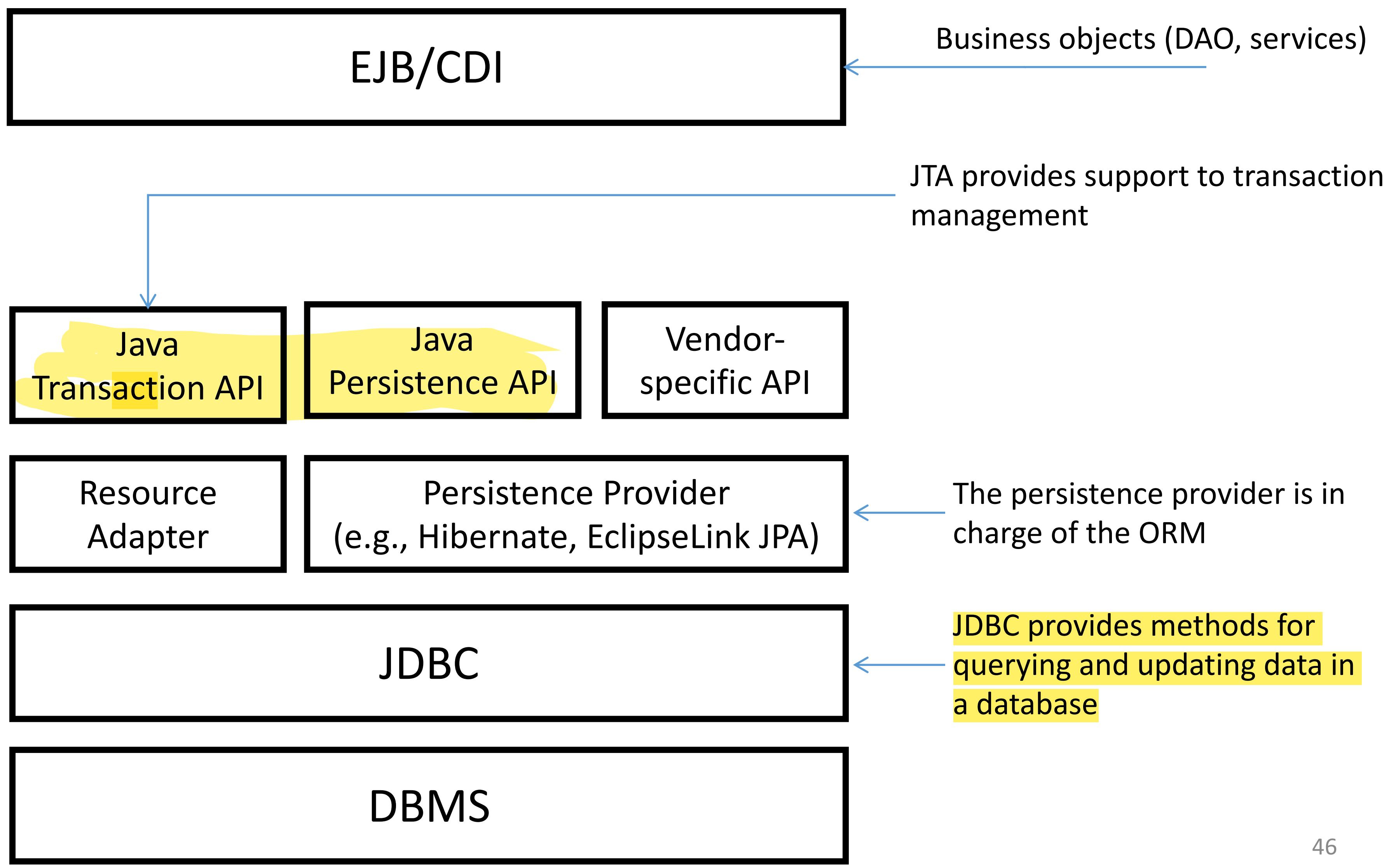
Differences between OM and RM

Object Oriented Model (Java)	Relational Model
Objects, classes	Tables, rows
Attributes, properties	Columns
Identity (physical memory address)	Primary key
Reference to other entity	Foreign key
Inheritance/Polymorphism	Not supported
Methods	Stored procedures, triggers
Code is portable	Not necessarily portable (depending on the vendor)

Java Persistence API

- The **Java Persistence API** bridges the gap between object-oriented domain models and relational database systems
  - JPA provides a POJO (Plain Old Java Object) persistence model for object-relational mapping
- Adds up to other previous proposals
  - JDO: Java data Objects (JDO 2.0: JSR 243)
  - JDBC: Java database connectivity (JDBC 3.0 API)
- Developed as part of JSR-317
  - In addition to support within EJB, JPA can be used in a standalone Java SE environment
  - Usable with / without a container

JPA Architecture


EJB/CDI
Business objects (DAO, services)
JTA provides support to transaction management
Java Transaction API
Java Persistence API
Vendor-specific API
Resource Adapter
Persistence Provider (e.g., Hibernate, EclipseLink JPA)
The persistence provider is in charge of the ORM
JDBC
JDBC provides methods for querying and updating data in a database
DBMS

JPA in a nutshell

- Java Persistence API main features:
  - **POJO Persistence:** there is nothing special about the objects being persisted, any existing non-`final` object with a default constructor can be persisted
  - **Non-intrusiveness:** the persistence API exists as a separate layer from the persistent objects, i.e., the mapped objects are not aware of the persistence layer
  - **Object queries:** a powerful query framework offers the ability to query across entities and their relationships without having to use concrete foreign keys or database columns

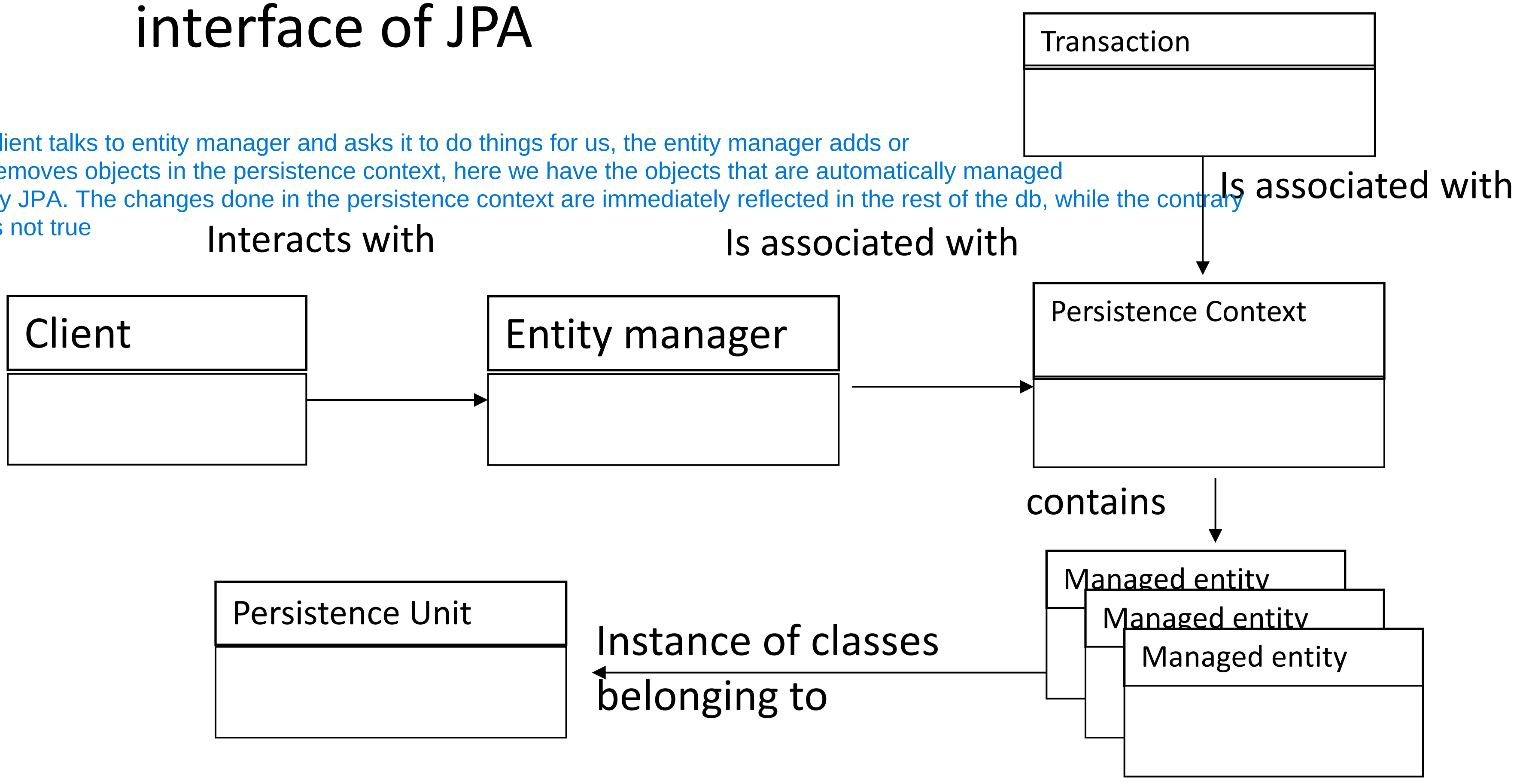
JPA main concepts

- Entity: a class (JavaBean) representing a collection of persistent objects mapped onto a relational table
- Persistence Unit: the set of all classes that are persistently mapped to one database (analogous to the notion of db schema)
- Persistence Context: the set of all managed objects of the entities defined in the persistence unit (analogous to the notion of db instance)
- Managed entity: an entity part of a persistence context for which the changes of the state are tracked
- Entity manager: the interface for interacting with a Persistence Context
- Client: a component that can interact with a Persistence Context, indirectly through an Entity Manager (e.g., an EJB component)

How to work with entities

- Entities are accessed through the Entity Manager interface of JPA

client talks to entity manager and asks it to do things for us, the entity manager adds or removes objects in the persistence context, here we have the objects that are automatically managed by JPA. The changes done in the persistence context are immediately reflected in the rest of the db, while the contrary is not true



Entity Manager interface

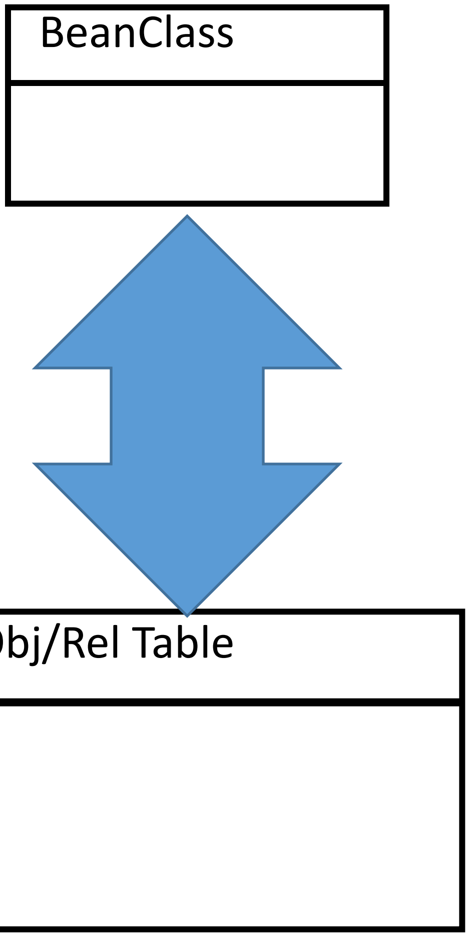
- The Entity Manager exposes all the operations needed to synchronize the managed entities in the persistence context to the database

Method signature	Description
`public void persist(Object entity);`	Persists an entity instance in the database
`public <T> T find(Class<T> entityClass, Object primaryKey);`	Finds an entity instance by its primary key
`public void remove(Object entity);`	Removes an entity instance from the database
`public void refresh(Object entity);`	Resets the entity instance from the database
`public void flush();`	Writes the state of entities to the database immediately

Entity

- A Java Bean (POJO: Plain Old Java Object) that gets associated with a tuple in a database
  - The class maps to the table
  - The objects map to the tuples
- The persistent counterpart of an entity has a life longer than that of the application
- The entity class must be associated with the database table it represents (mapping)
- An entity **can** enter in a managed state, where all the modifications to the object’s state are tracked and automatically synchronized to the database

Entity Properties



- Identification (primary key)
- Nesting
- Relationship
- Referential integrity (foreign key)
- Inheritance

Entity (example)

Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    public Employee() {}
    public Employee(int id) { this.id = id; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary (long salary) { this.salary = salary; }
}
```

Java annotations are used to qualify the class as an entity

The entity instance is just a POJO

Entity constraints

- Entities must respect the following requirements:
    - The entity class must have a `public` or `protected` no-arg constructor
    - The entity class must not be `final`
    - No method or persistent instance variables of the entity class may be final
    - If an entity instance is to be passed by value as a detached object, the `Serializable` interface must be implemented
- The persistent state of an entity is represented by instance variables, which correspond to JavaBean properties (i.e., with setter and getter methods)

Entity Identification

- In the database, objects and tuples have an identity (primary key)
- → an entity assumes the identity of the persistent data it is associated with
- Simple Primary key = persistent field of the bean used to represent its identity
- Composite Primary key = set of persistent fields used to represent its identity
- Remark: with respect to the POJOs, the persistent identity is a new concept. **POJOs do not have a durable identity**

Entity identification syntax

```
@Entity
public class Mission
  implements Serializable {

  @Id
  private int id;
  private String city;
..
```

- `@Id` tags a field as the simple primary key
- Composite primary keys are denoted using the `@EmbeddedId` and `@IdClass` annotations

Entity: Identifier generation

- Sometimes, applications do not want to explicitly manage uniqueness of data values
    - The persistence provider can automatically generate an identifier for every entity instance of a given type
- This persistence provider’s feature is called **identifier generation** and is specified by the `@GeneratedValue` annotation

Identifier generation options

- Applications can choose one of four different ID generation strategy

Id generation strategy	Description
`AUTO`	The provider generates identifiers by using whatever strategy it wants
`TABLE`	Identifiers are generated according to a generator table
`SEQUENCE`	If the underlying DB supports sequences, the provider will use this feature for generating IDs
`IDENTITY`	If the underlying DB supports primary key identity columns, the provider will use this feature for generating IDs

Identifier generation annotation

```
@Entity
public class Mission implements Serializable {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;

private String city;
```

- Whenever possible let the database generate the unique Ids

Attribute specifications

- Attributes can be qualified with properties that direct the mapping between POJOs and relational tables
- Mapping base and special types
  - Large objects
  - Enumerated types
    - As Java enumerations
    - As strings
  - Temporal types
    - Java temporal types
    - JDBC temporal types
- Specifying fetch policy
  - LAZY
  - EAGER (default)

```
@Entity
public class Mission implements
   Serializable {
   @Id
   @GeneratedValue(strategy =
   GenerationType.IDENTITY)
   private int id;
   @Temporal(TemporalType.
   DATE)
   private Date date;
   private MissionStatus status;
    @Basic(fetch=FetchType.LAZY)
    @LOB
    private byte[] photo;
}
```

- Specifying LAZY policy makes the attribute value remain empty when the object is retrieved, until the attribute is accessed explicitly (→ use for LOBs only)

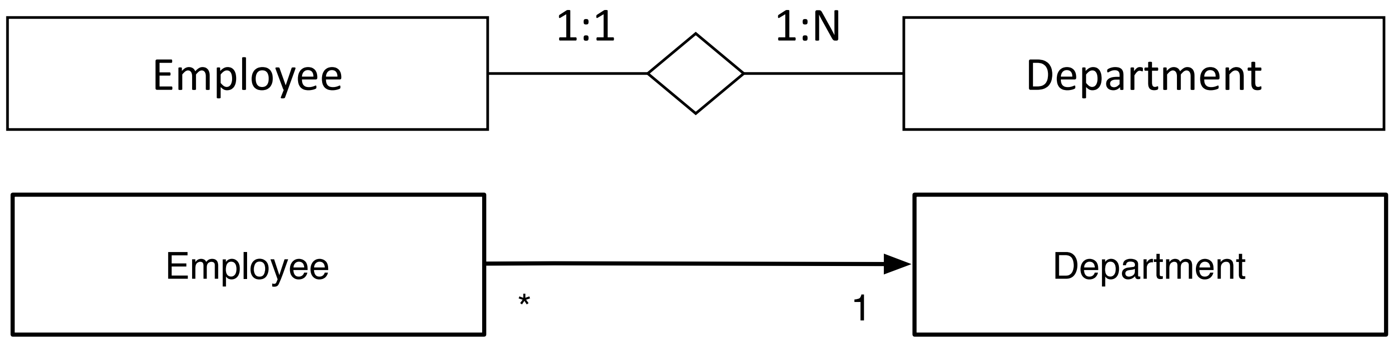
Object to table mapping

- By default entities are mapped to tables with the same name and their fields to columns with the same names
- Defaults can be overridden
- Some annotations (e.g., `@column`) may have attributes used for generating the database schema from the entity classes (e.g., `nullable`)
- Entities may also have attributes that are not persisted (`@Transient`)

```
@Entity @Table(name="T_BOOKS")
public class Book {

 @Column(name="BOOK_TITLE",
  nullable=false)
  private String title;
  private CoverType coverType;
  private Date publicationDate;
  @Transient
  private BigDecimal discount;
}
```

Entities and relationships

- If entities contained only simple, persistent states, the issue of ORM would be a trivial one
- In fact, most entities need to be able to have **relationships** with other entities
- NOTE: there is an ambiguity in the meaning of "relationship" in the Entity-Relationship conceptual model and in the Object Model
- From now on we will use relationship in the sense of the Object Model



Relationship concepts

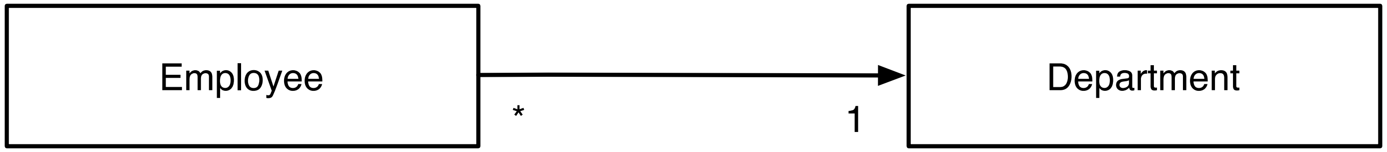
- Every (OM) relationship has four characteristics:
  - **Directionality**: each of the two entities may have an attribute that enables access to the other one
  - **Role**: each entity in the relationship is said to play a role with respect to one direction of access
  - **Cardinality**: the number of entity instances that exist on each side of the relationship
  - **Ownership**: one of the two entity in the relationship is said to <u>own</u> the relationship

Directionality

- Each entity in the relationship may have a **reference** to the other entity:
    - When each entity refers to the other, the relationship is **bidirectional**
    - If only one entity has a reference to the other, the relationship is said to be **unidirectional**
    - The reference can be single-valued or set-valued (collection of references)
- All relationships in JPA are unidirectional
    - A bidirectional relationship is intended as a “matched” pair of unidirectional mappings
    - “Matching” must be declared explicitly (via the dedicated attribute `mappedBy`)

Roles

- Based on directionality, one entity plays the role of **source** and one the role of **target**
- In the relationship from `Employee` to `Department`
  - `Employee` is the source entity
  - `Department` is the target entity

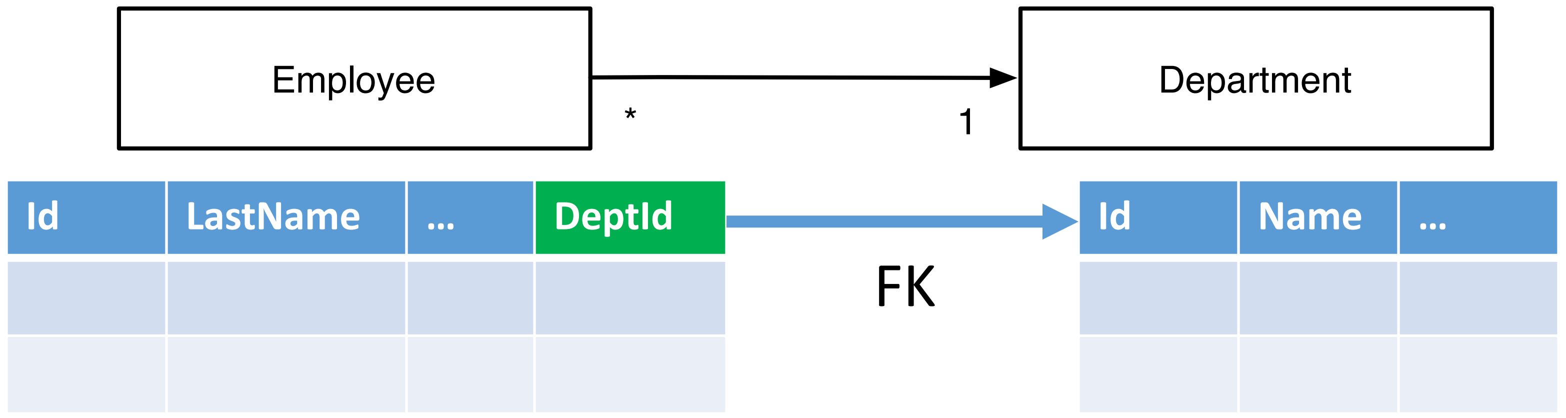


Cardinality

- Each role in the relationship has its own cardinality. This leads to four possible combinations:
  - **Many-to-one:** many source entities, one target entity
  - **One-to-many:** one source entity, many target entities
  - **One-to-one:** one source entity, one target entity
  - **Many-to-many:** many source entities, many target entities
- Remember: bidirectional relationships (the E-R way) are just pairs of matched unidirectional relationships with swapped source and target entities
- A to-one relationship implies that the source entity has **one reference** to the target entity,
- A to-many relationship implies that the source entity has a **multiple references** (collection) to the target entity

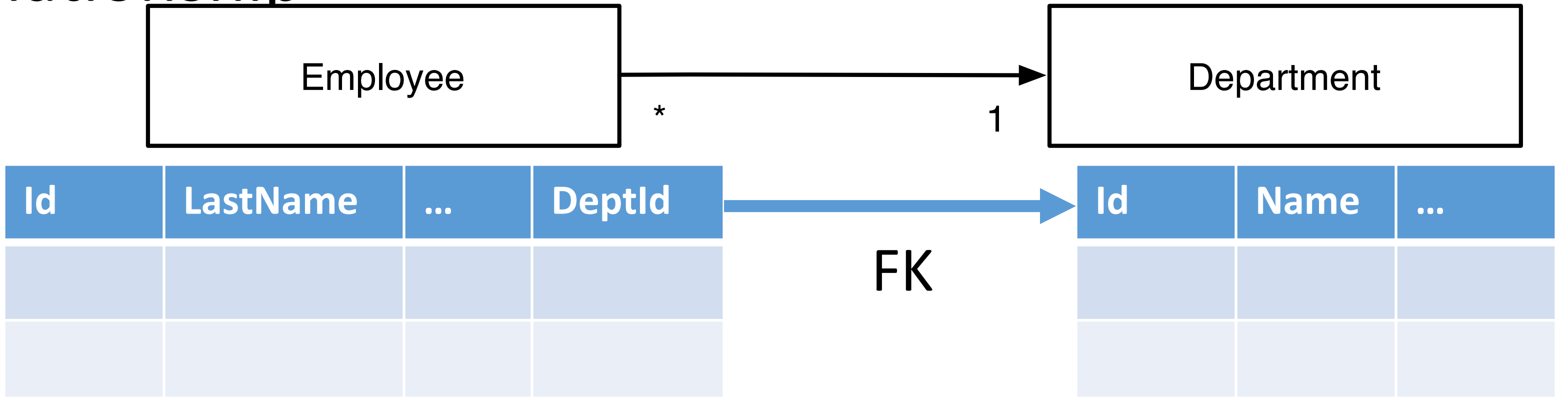
Ownership (1/2)

- In the database, relationships are implemented by a foreign key column that refers to the key of the referenced table
  - In JPA, such a column is called **join column**
- Example:
  - Many-to-one unidirectional relationship between Employee and Department
  - The underlying Employee table has a FK column containing the Department primary key



Ownership (2/2)

- In 1:N and 1:1 relationships, one of the two entities will have the FK column in its table
  - This entity is called the **owner** of the relationship and its side is called the **owning side**
- Example: Employee is the owner of the relationship
- Ownership is important because the annotations that define the physical mapping of the relationship (FK column) are specified in the owner side of the relationship



Possible relationship mappings

- 1:N
  - Bidirectional
  - Unidirectional one-to-many & many-to-one
- 1:1
  - Bidirectional
  - Unidirectional
- N:M
  - Bidirectional
  - Unidirectional
- Whether to go for uni- or bi-directionality depends on the application
- If the relationship is accessed from one entity to the other and vice versa, use bi-directionality otherwise use uni-directionality
- NOTEs:
  - Bi-directionality can be also implemented with unidirectional relationship mapping + set-valued query
  - Performance considerations may prompt for the definition of bidirectional mappings even if only one direction is used

1:N bidirectional

- Expressed with (`@ManyToOne` + `@OneToMany` + `mappedBy`)
- The many-to-one mapping direction is defined by annotating the entity that participates with multiple instances with the `@ManyToOne` annotation
- The entity that contains the `@ManyToOne` annotation is the **owner** of the relationship because its underlying table stores the FK to the referenced table
- The `@JoinColumn` annotation can be used to specify the FK column of underlying table (defaults to the data member name)

@ManyToOne annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToOne
    @JoinColumn(name="dept_fk") --> name on the database of the column acting as foreign key
    private Department dept;
    ...
}
```

1:N bidirectional, continued

- To achieve bi-directionality, the one-to-many mapping direction must be specified too
- This is done by including a `@OneToMany` annotation in the entity that participates with one instance
- The `@OneToMany` annotation is placed on a **collection** data member and comprises a `mappedBy` element to indicate the property that implements the inverse of the relationship
- NOTE: In a one-to-many mapping the owner of the relationship is still the entity that participates with multiple instances

`@OneToMany` annotation in `Department.java`

```
@Entity
public class Department {
     @Id private int id;
     @OneToMany(mappedBy="dept")
     private Collection<Employee> employees;
     ...
}
```

The attribute on the target entity that implements the inverse of the relationship

Uni-directional 1:N: many-to-one

- Sometimes applications require the access to relationships only along one direction
- In this case the bidirectional mapping is not necessary
- The many-to-one direction case is trivial
- Use just the same annotations as in the bidirectional case and skip the inverse mapping

```
@Entity
public class Employee {
         @Id private int id;

         @ManyToOne
         @JoinColumn(name="dept_fk")
         private Department department;
         ...
}
```

Uni-directional 1:N: one-to-many

- The one-to-many direction is less trivial
- Two alternatives are viable performance-wise
  - Map the relationship as in the bi-directional case and use only the one-to-many direction
  - Do not map the collection attribute in the entity that participates with one instance and use a query instead to retrieve the correlated instances, relying on the inverse (many-to-one) relationship direction mapping

```
List<Employee> emps =
entityManager.createQuery(
  "SELECT e " +
  "FROM Employee e " +
  "WHERE e.dept.id = :deptId",
  Employee.class)
.setParameter("deptId", "DEIB")
.getResultList();
```

`@joincolumn` vs `mappedBy`

- The annotation `@JoinColumn` indicates the FK column that implements the relationship in the database; such annotation is <u>normally</u> inserted in the entity owner of the relationship (i.e., the one mapped to the table that actually contains the FK column)
  - Used to drive the generation of the SQL code to extract the correlated instances
- The `mappedBy` attribute indicates that “this side” is the inverse of the relationship, and the owner resides in the "other" related entity
  - Used to specify bidirectional relationships

Why specifying `MappedBy`?

- In absence of the `mappedBy` parameter the default JPA mapping (created when the database is generated from the JPA entities) uses a bridge table (as for N:M relationships)
- The purpose of the `mappedBy` parameter is to instruct **JPA** NOT to create a bridge table as the relationship is already being **mapped by** a FK in the opposite entity of the relationship

One-to-one mapping (1/2)

- In a one-to-one mapping the owner can be either entity, depending on the database design
- A one-to-one mapping is defined by annotating the owner entity (the one with the FK column) with the `@OneToOne` annotation

@OneToOne annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne
    private ParkingSpace parkingSpace;
    ...
}
```

One-to-one mapping (2/2)

- If the one-to-one mapping is bidirectional, the inverse side of the relationship needs to be specified too
- In the other non-owner entity, the `@OneToOne` annotations must come with the `mappedBy` element
- The presence of `mappedBy` tells JPA that the foreign key constraint is in the table mapping the OTHER entity

The one with mapby is the owner side

@OneToOne annotation (inverse side) in ParkingSpace.java

```
@Entity
public class ParkingSpace {
    @Id private int id;
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    ...
}
```

Many-to-many mappings (1/2)

- In a many-to-many mapping there is no FK column
  - Such a mapping is implemented by means of a **join table** (aka **bridge table**)
- Therefore, we can arbitrarily specify as owner either entity

@ManyToMany annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToMany
    private Collection<Project> projects;
        ...
}
```

Many-to-many mappings (2/2)

- If the many-to-many mapping is bidirectional, the inverse side of the relationship needs to be specified too
- In the other entity, the `@ManyToMany` annotation must come with the `mappedBy` element

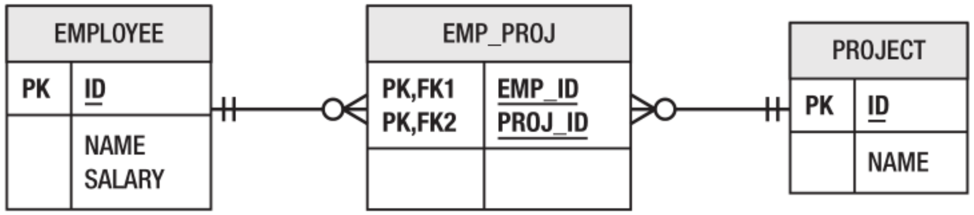
@ManyToMany annotation (inverse side) in Project.java

```
@Entity
public class Project {
    @Id private int id;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    ...
}
```

Join Table

- The logical model of a N:M relationships requires a bridge table (JOIN table in the JPA terminology)
- The non-default mapping of the entity to the bridge table is specified via annotation

```
@Entity
public class Employee {
 @Id private long id;
 private String name;
 @ManyToMany
 @JoinTable(name="EMP_PROJ",
  joinColumns=@JoinColumn(name="EMP_ID"),
  inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
 private Collection<Project> projects;
// ...
}
```



- When no @JoinTable is specified, then a default name `<Owner>_<Inverse>` is assumed, where `<Owner>` is the name of the owning entity, and `<Inverse>` is the name of the inverse or non-owning entity (`Employee_Project` in the example)

Prima di questa slide ha fatto un riassunto delle cose da sapere bene

Relationship fetch mode (1/3)

- **When loading** an entity, it is questionable if related entities are to be fetched & loaded too
  - Performance can be optimized by deferring data fetch until the time when they are needed
- This design pattern is called **lazy** (opposite= **eager**)
- Loading policy can be expressed specifying **fetch mode** for relationships (as seen for attributes of LOB type)
- At relationship level, lazy loading can greatly enhance performance because it can reduce the amount of SQL that is executed if correlated instances are seldom accessed by the application

Relationship fetch mode (2/3)

- When the fetch mode is not specified, by default:
  - A single-valued relationship is fetched **eagerly**
  - Collection-valued relationships are loaded **lazily**
- In case of bidirectional relationships, the fetch mode might be lazy on one side but eager on the other
  - Quite common situation, relationships are often accessed in different ways depending on the direction from which navigation occurs
- NOTE: Consider lazy loading as the most appropriate mode for all relationships, because if an entity has **many** single-valued relationships that are **not all used by applications**, the default eager mode may incur performance penalties

Lazy loading of the parkingSpace attribute

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    ...
}
```

Relationship fetch mode (3/3)

- The directive to lazily fetch an attribute is meant only to be a **hint** to the persistence provider
  - The provider is not required to respect the request because the behavior of the entity is not compromised if the provider decides to eagerly load
- The converse is **not true** because specifying that an attribute be eagerly fetched might be critical to access the entity once detached (i.e., no longer managed)

Cascading operations (1/4)

- By default, every Entity Manager operation applies **only to the entity supplied as an argument to the operation**
  - The operation **will not** cascade to other entities that have a relationship with the entity that is being operated on
- For some operations (e.g., `remove()`) this is <u>usually</u> the **desired behavior**

Cascading operations (2/4)

- Some other operations usually require cascading, such as `persist()`
  - If an entity has a relationship to another (dependent, aka "child") entity, normally the child entity must be persisted together with the parent entity
- Example: Many-to-one unidirectional mapping between `Employee` and `Address`

Manually cascading

```
Employee emp = new Employee();
Address addr = new Address();
emp.setAddress(addr);
em.persist(addr);
em.persist(emp);
```

We would like to avoid explicit persisting the `Address` entity instance

Cascading operations (3/4)

- The `cascade` attribute is used to define when operations should be automatically cascaded across relationships
    - As the entity manager adds the `Employee` instance to the persistence context, it navigates the address relationship looking for a new `Address` entity to manage as well
    - The `Address` instance must be set on the Employee instance before invoking `persist()` on the employee object
    - If an `Address` instance has been set on the `Employee` instance and not persisted explicitly or implicitly via cascading, an error occurs

Enabling cascade persist

```
@Entity
public class Employee {
    @ManyToOne(cascade=CascadeType.PERSIST)
    Address address;
}
```

Cascading operations (4/4)

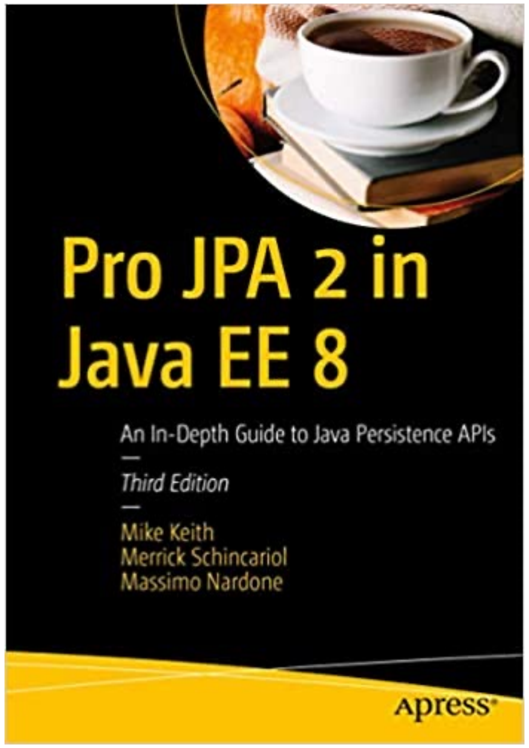
- The `cascade` attribute accepts several possible values specified in the `CascadeType` enumeration:
  - `PERSIST`, `REFRESH`, `REMOVE`, `MERGE` and `DETACH`
  - `ALL` is a shorthand for declaring that all five operations should be cascaded
- As for relationships, cascade settings are unidirectional
  - they must be explicitly set on both sides of a relationship, if the default behavior has to be overridden
- NOTE: there is no **default cascade type in JPA**.
  By **default** no operations are cascaded

Orphan removal

- JPA supports an additional remove cascading mode specified using `orphanRemoval` in `@OneToOne` and `@OneToMany`
  - appropriate for **privately owned** parent-child relationships, in which every child entity is associated only with one parent entity through just one relationship (weak entity in the E-R terminology)
  - causes the child entity to be removed when the parent-child relationship is broken either by removing the parent or by setting to null the attribute that holds the related entity, or in the one-to-many case, by removing the child entity from the parent's collection
- Difference between `CascadeType.REMOVE` and `orphanRemoval=true`
  - For orphan removal: invoking `setEmployees(null)` on `Department` the related `Employee` entities are removed from the database automatically
  - For remove cascade: invoking `setEmployees(null)` on `Department`, the related `Employee` entities are **NOT** removed from the database automatically
  - If the orphan object is not in the managed state, removal does not occur

References

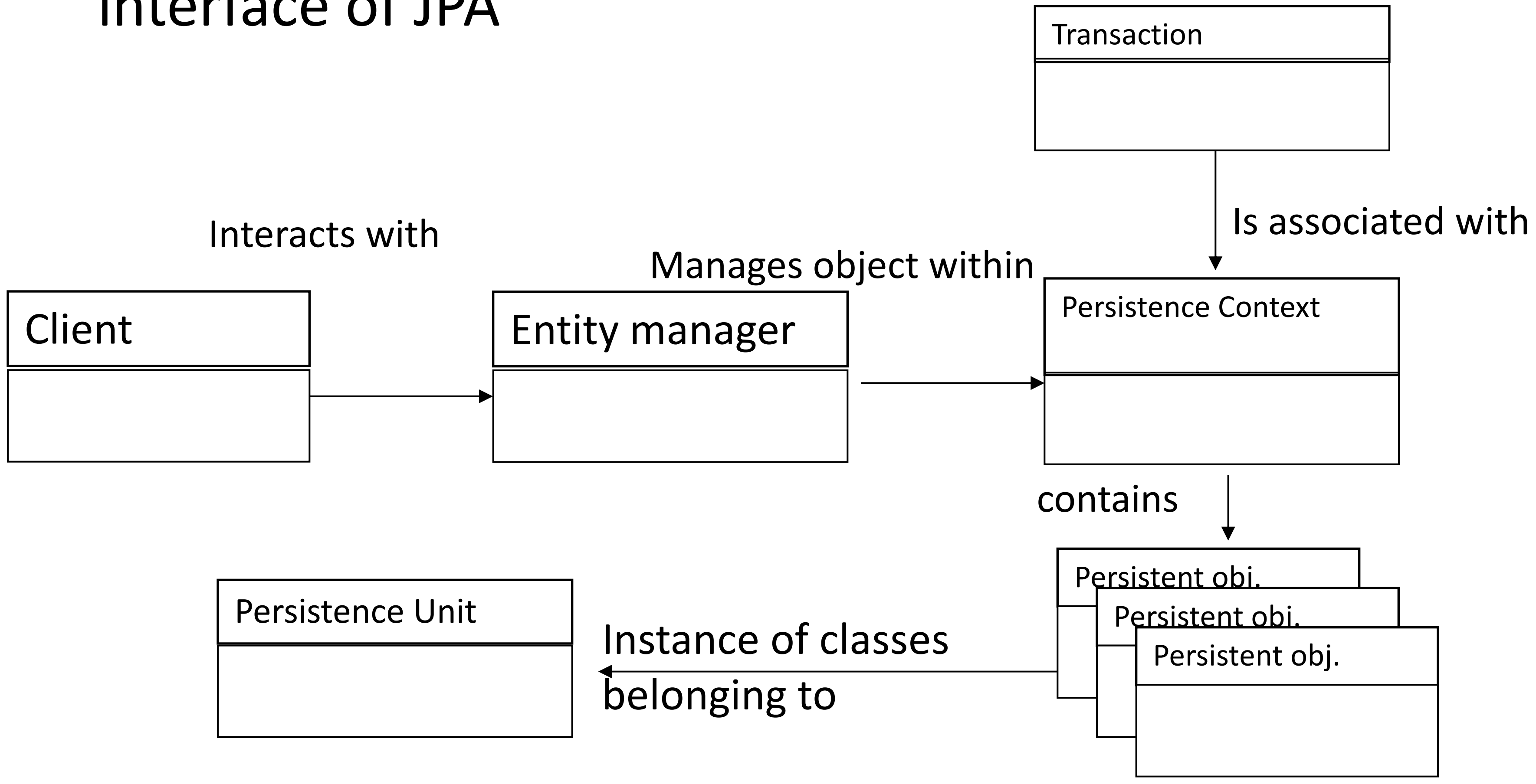
- JPA specifications, http://www.jcp.org/en/jsr/detail?id=317
- Pro JPA 2 in Java EE8, *M. Keith, M. Schincariol, Apress Media LLC*
- *Chapter 4 and 10*



Java Persistence API
The Entity Manager

How to work with entities

- Entities are accessed through the Entity Manager interface of JPA



EntityManager

- Because entity instances are plain Java objects, they **do not become managed until the application invokes an API method to initiate the process**
- The Entity Manager is the central authority for all persistence actions
  - It manages the mapping between a fixed set of entity classes and an underlying data source
  - It provides APIs for creating queries, finding objects, and synchronizing objects and database state

Persistence Context

- It is the **fundamental (and somehow elusive) concept of JPA:** a kind of main memory database that holds the objects in the managed state
- A managed object is **tracked**, i.e., all the modifications to its state are monitored for **automatic alignment to the database**
- Database writes by default occur **asynchronously**, at a time decided by the persistence provider implementation
- For the database writes to happen, the **Persistence Context must hook up to a transaction**, which is the only means to write to the database
- A managed entity has **two lives**: one as a Java object and one as a relational tuple bound to it (ID of the POJO = PK of the tuple)
- Such a binding exists **ONLY inside the persistence context.** When the POJO exits the persistence context the binding breaks: it gets untracked and no longer synchronized to the database
- NOTE: the application **never sees the persistence context**, it interacts ONLY with the Entity Manager

`EntityManager` Interface

Method signature	Description
`public void persist(Object entity);`	Makes an entity instance become part of the persistence context, i.e., managed
`public <T> T find(Class<T> entityClass, Object primaryKey);`	Finds an entity instance by its primary key
`public void remove(Object entity);`	Removes an entity instance from the persistence context and thus from the database
`public void refresh(Object entity);`	Resets the state of entity instance from the content of the database
`public void flush();`	Writes the state of entities to the database as immediately as possible

Creating a new POJO

- Calling the `new` operator **does not** magically interact with some underlying service to create the `Employee` in the database
  - Instances of the `Employee` class remain POJOs until the application asks the `EntityManager` to start managing the entity
- When an entity is first instantiated, it is in the **transient** (or **new**) state since the `EntityManager` **does not know it exists yet**
- Transient entities are not part of the persistence context associated with the Entity Manager

Creating a new POJO

```
Employee emp = new Employee(ID, "John Doe”); //emp unknown to the EM!
```

Making an entity managed

- Entity Manager's `persist()` method makes a transient entity become **managed** **(BIG WARNING: managed <> written to the DB!)**
- A managed entity "lives" in a persistence context and the `EntityManager` makes sure that any change to its state is tracked for being persisted to the database, **sooner or later**
- When the transaction associated with the Entity Manager's persistence context commits a new record is created in the database mapping the entity
- The Entity Manager `flush()` method can be called to ask the Persistence provider to write changes as soon as possible to the database
- The managed entity and the corresponding tuple become associated until the entity exits the managed state: changes to the entity will be reflected to changes to the tuple
- NOTE: calling `persist()` on an already managed entity instance is possible and triggers the cascade process

Persisting an entity

```
Employee emp = new Employee(ID, "John Doe");
em.persist(emp); // emp now is managed
```

Finding an entity

- `EntityManager`’s `find()` method takes as an input the class of the entity that is being sought and the primary key value that identifies the desired entity instance
- When the call completes, the returned object will be **managed**, added to the persistence context
    - If the entity instance is not found, then the `find()` method returns `null`
- The actual amount of data extracted and added to the persistence context depends on the **fetch policy** of attributes and relationships

Finding an entity

```
Employee emp = em.find(Employee.class, ID); //emp is managed
```

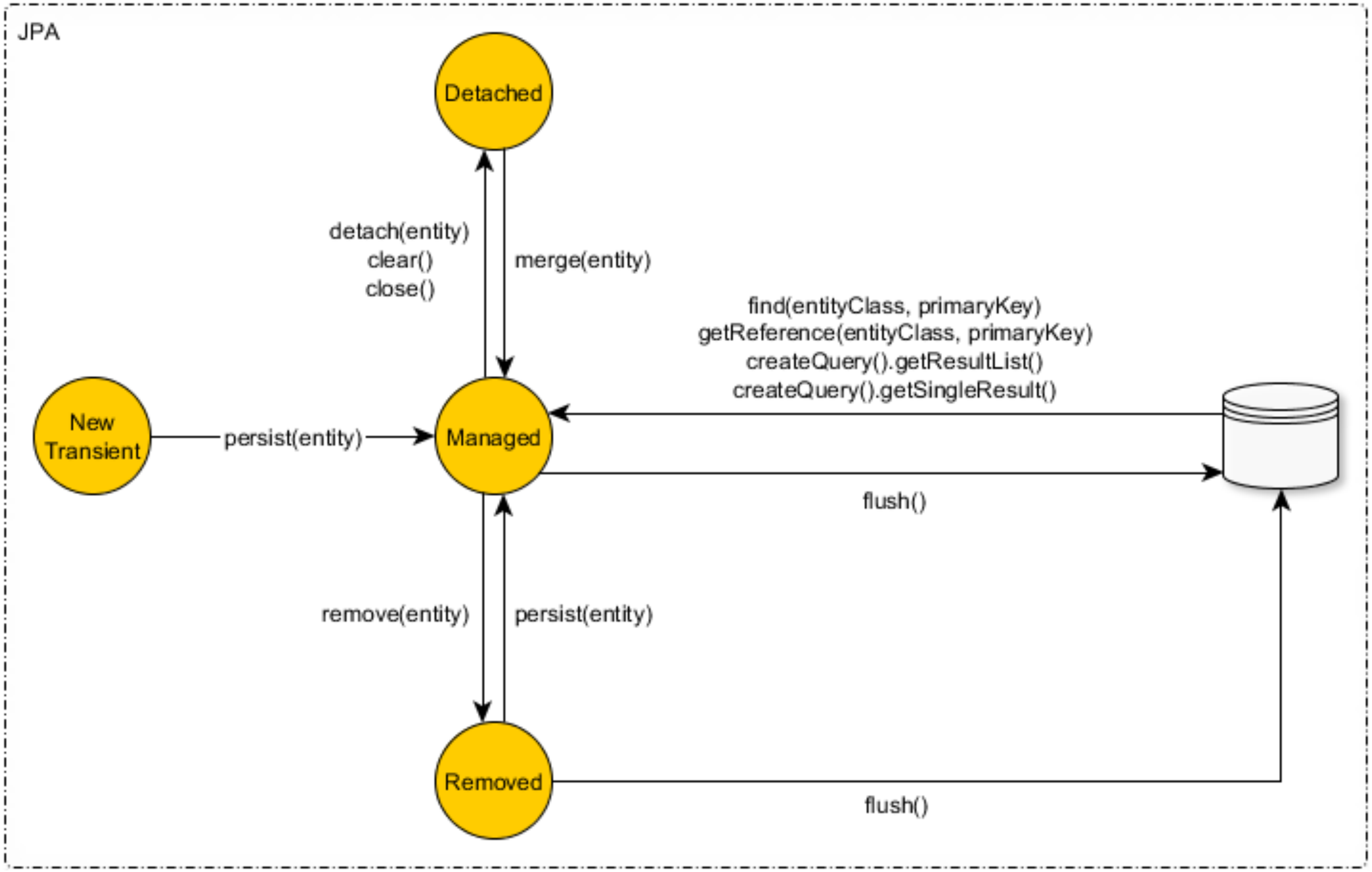
Removing an entity

- Entity Manager’s `remove()` method breaks the association between the entity and the persistence context
- When the transaction associated with the Entity Manager’s persistence context commits or the Entity Manager `flush()` method is called, the tuple associated with the entity is scheduled for deletion from the database
- The entity (Java object) still exists but its changes are no longer tracked for being (later) synchronized to the database

Removing an entity

```
em.remove(emp); // emp removed, no longer managed
```

EM operations & entity state transitions



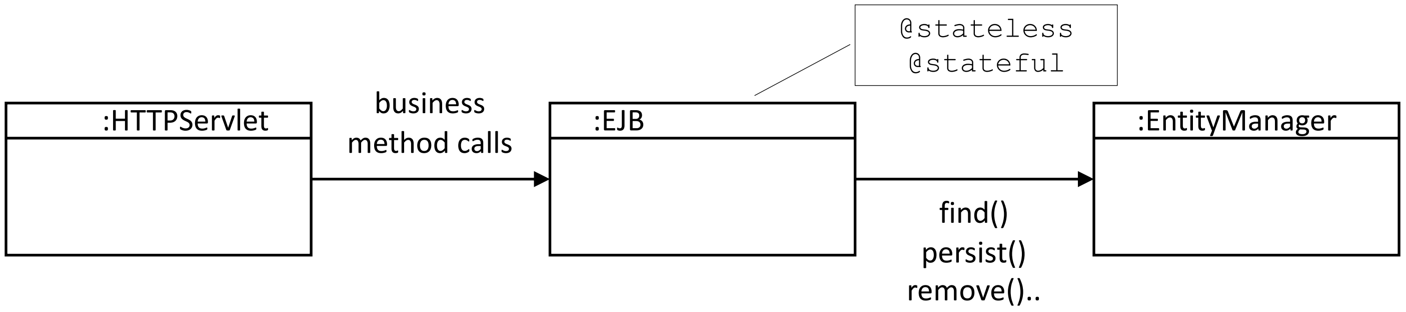
Credits https://vladmihalcea.com/a-beginners-guide-to-jpa-hibernate-entity-state-transitions/

Entity lifecycle legenda

- NEW: Unknown to the Entity Manager, no persistent identity, no tuple associated
- MANAGED: Associated with persistence context, changes to objects automatically synch to database (**NOT VICE VERSA!**)
- DETACHED: Has an identity potentially associated with a database tuple but changes are NOT automatically propagated to database
- REMOVED: Scheduled for removal from the database
- DELETED: erased from the database

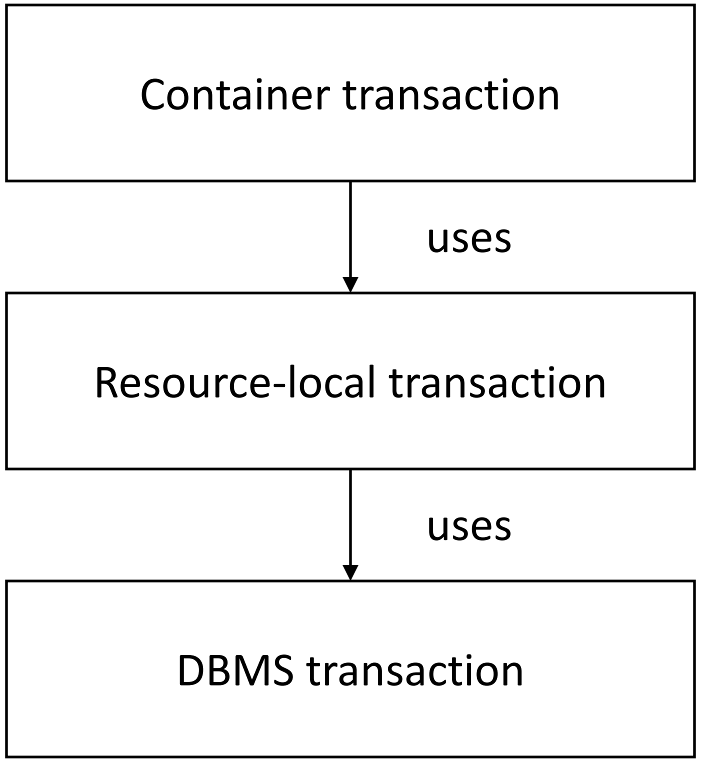
JPA application architectures

- JEE
  - The client exploits the services of a container (EJB or CDI) to connect to the Entity Manager
    - The client (e.g., a Web servlet) interacts with a business object (e.g., an EJB or a CDI component)
    - The business component interacts with the Entity Manager
  - The container provides the support to make JPA entity method calls **transactional** through the automatic creation of transactions



Transaction management in JPA

- Database application development requires understanding the transactional properties of code
- How the Entity Manager participates in transactions
- When a transaction starts/ends
- Even if transactions are managed transparently by the container, understanding how they work is necessary to predict application behaviour
- Transactions exist at three levels
    - DBMS transactions
    - Resource-local transactions
    - Container transaction



Transaction abstraction levels

- DBMS
  - They live inside the DBMS are demarcated by SQL commands and managed by the DBMS
- Resource-local
  - They are created and demarcated by means of JDBC commands issued through the JDBC connection interface. They are mapped by the JDBC driver to DBMS transactions. **They must be managed by the application**
- Container (we will use this level)
  - They are defined through the JTA interface and mapped by the JTA Transaction Manager and EJB container to JDBC transactions. **They can be managed by the application or by the container (EJB, servlet or CDI)**

CM Entity Manager

- Container Managed

```
@Stateless // this is a business object (EJB)
public class myEJBService {
  @PersistenceContext(unitName = "MyPersistenceUnit")
  private EntityManager em;
```

- The container injects the EM instance into the business object
- The container creates and destroys instances of the Entity Manager transparently to the application
- This is the EM type **used by default**, if no other specification is added
- The container **provides the transaction** needed for saving the modifications made to the entities of the Persistence Context associated with the Entity Manager into the database

CM transaction: example

- (Web) client code (e.g., in a servlet)

```
@EJB(name = "it.polimi.db2.mission.services/MissionService")
private MissionService mService; // client obtains a business object
. . .
Mission mission = mService.closeMission(missionId, userId);
```

- Business component code MissionService EJB

```
// transaction started here by the container
public void closeMission(int missionId, int reporterId) throws
                        BadMissionReporter, BadMissionForClosing {

 // persistence context created and associated with transaction
 Mission mission = em.find(Mission.class, missionId);
 if (mission.getReporter().getId() != reporterId) {
                                        throw new BadMissionReporter(" . . .");
 }
 if (mission.getStatus() != MissionStatus.REPORTED) {
             throw new BadMissionForClosing(" . . . ");
 }
 mission.setStatus(MissionStatus.CLOSED);

} // transaction committed by container
  // no explicit transaction demarcation code needed!
```

Transaction and method calls

- When a client (e.g., a Web Servlet) calls a method of a business object that exploits a (default) CM EM for persistence, the (EJB) container provides (creates or reuses) a transaction for saving the modifications to the database
- If the called method in turns calls another method of the same or of a different business object, the same transaction is reused
- This is the most common and default behavior, but the business objects methods can be annotated to specify a different way to use the transactions provided by the container

Transactional behaviour of methods

- When **Container Managed** transactions are used the container wraps the method calls of managed components and execute them in a transaction
- A method can be annotated to obtain the desired transactional behaviour with `@TransactionAttribute(TransactionAttributeType.XXX)`
- Where XXX can be:
  - MANDATORY: a transaction is expected to have already been started and be active when the method is called. If no transaction is active, an exception is thrown
  - **REQUIRED: the default, if no transaction is active one is started. If one is active (e.g., created by the caller) this is used**
  - REQUIRES_NEW: the method always needs to be in its own transaction. Any active transaction is suspended
  - SUPPORTS: the method does not access transactional resources, but tolerates running inside one if it exists
  - NOT_SUPPORTED: the method will cause the container to suspend the current transaction if one is active when the method is called
  - NEVER: the method will cause the container to throw an exception if a transaction is active when the method is called.

Benefits of JPA: summary

- No need to write SQL code
- Application code ignores table names
  - Mapping is expressed via annotations with defaults
- No need to create/destroy the connection
- No need to create and terminate the transaction (begin, commit, rollback managed by the container)
- Business objects methods automatically made transactional (“all or nothing” semantics at the method call level and not at the SQL statement execution level)
- Default transactional behaviour, with annotations to specify different transaction management policies