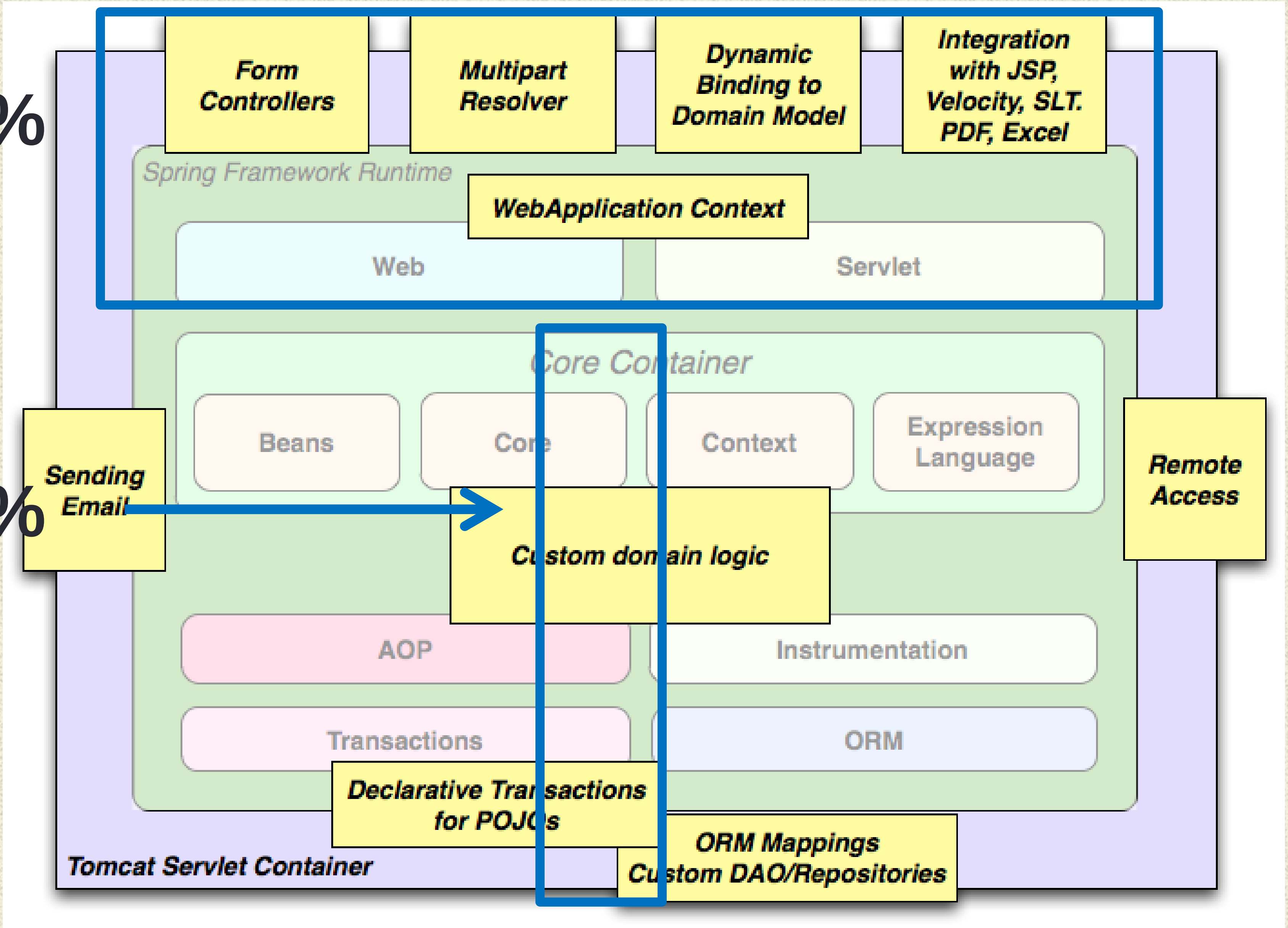
LESSON 12
PERSISTENCE & TRANSACTIONS

TAPPING THE SOURCE OF PURE KNOWLEDGE

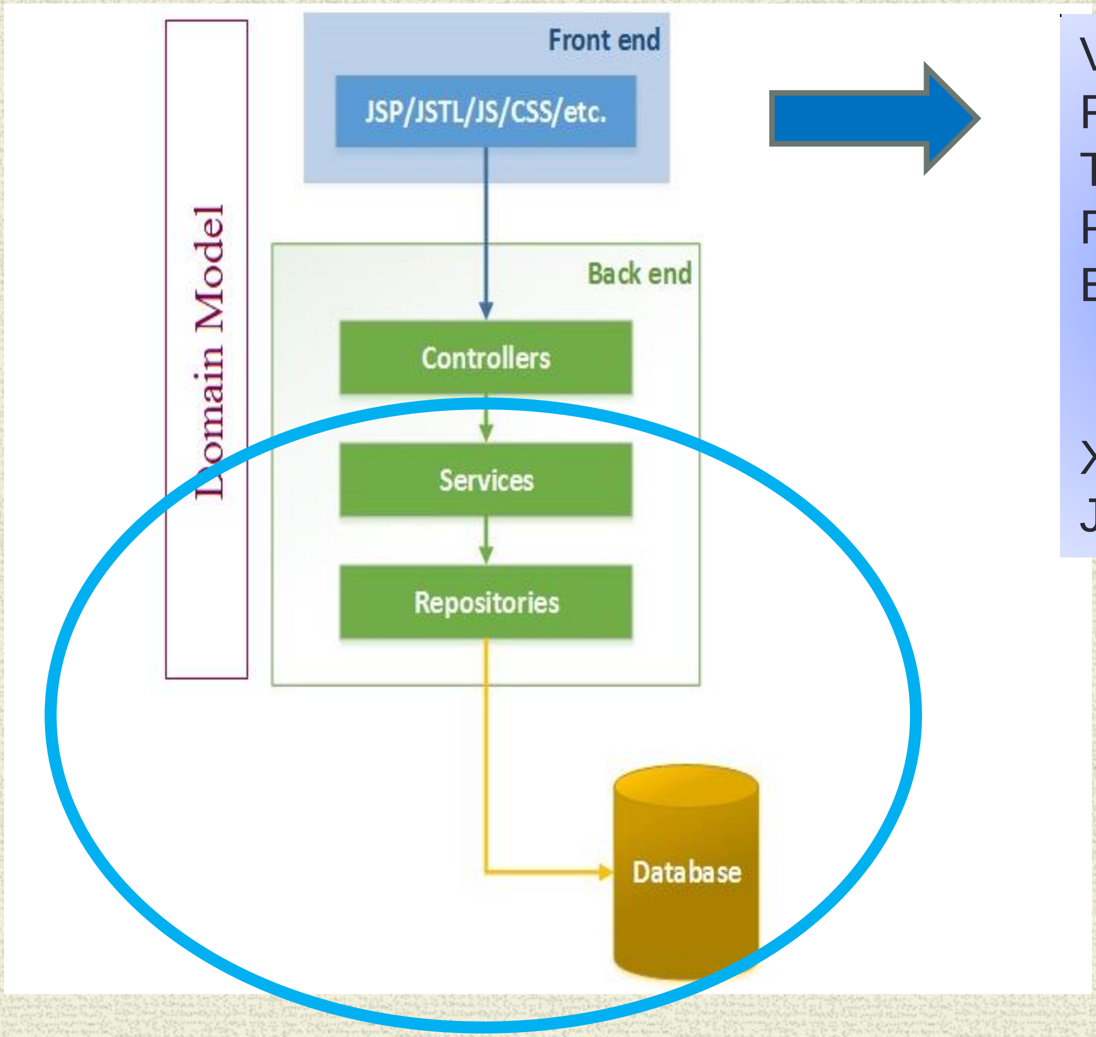
Spring N-tier Architecture

85%

15%

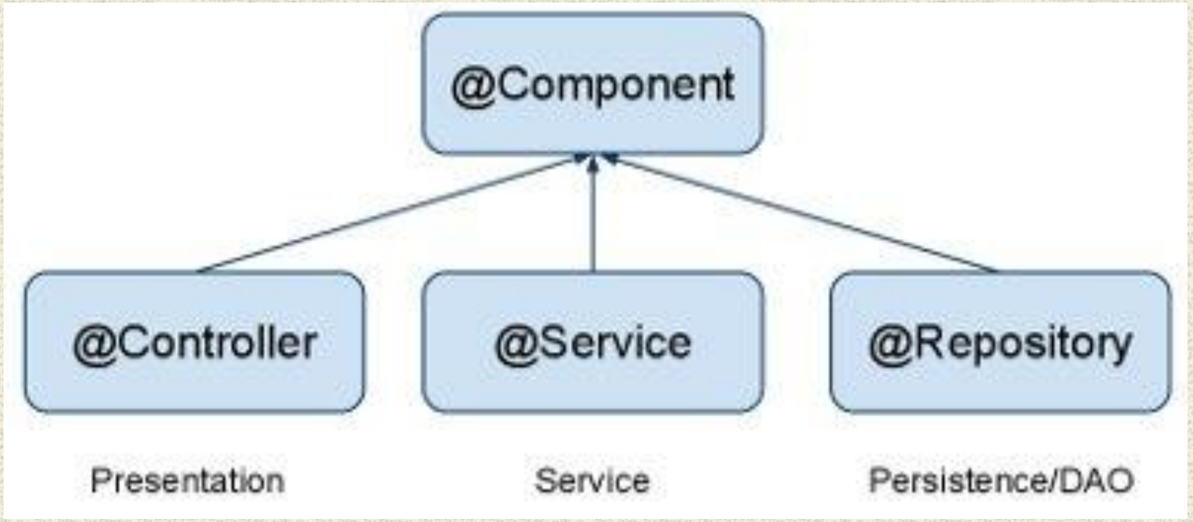


Spring MVC “Full Stack”



Velocity
FileMarker
ThymeLeaf
PDF
Excel
Web Flow
JSF
XML
JSON

Backend Components



@Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.

N-Tier Architecture

- Layered
  - Discrete responsibilities for different layers
  - Separation of Concerns

Presentation [View & Controller] Tier

Business Tier [Services]

Persistence Tier [Repository]

Microsoft N-tier

Service Tier “manages” Persistence

All access to Persistence through Services

Services responsible for business Logic

and data model composition

Business logic does NOT belong in Persistence

Business logic does NOT belong in Presentation

Spring/JPA/Persistence is designed with this architecture

Java Persistence API

JPA is a specification – not an implementation.

JPA 1.0 (2006). JPA 2.0 (2009).

Standardizes interface across industry platforms

Object/Relational Mapping

- **Specifically Persistence for RDBMS**

Major Implementations [since 2006]:

- Toplink - Oracle implementation [donated to Eclipse foundation for merge with Eclipselink 2008]
- Hibernate - Most deployed framework. Major contributor to JPA specification.
- OpenJPA - (openjpa.apache.org) which is an extension of Kodo implementation.

Spring Data

Spring Data

High level SpringSource project whose purpose is to unify and ease the access to different kinds of persistence stores, both relational database systems and NoSQL data stores.

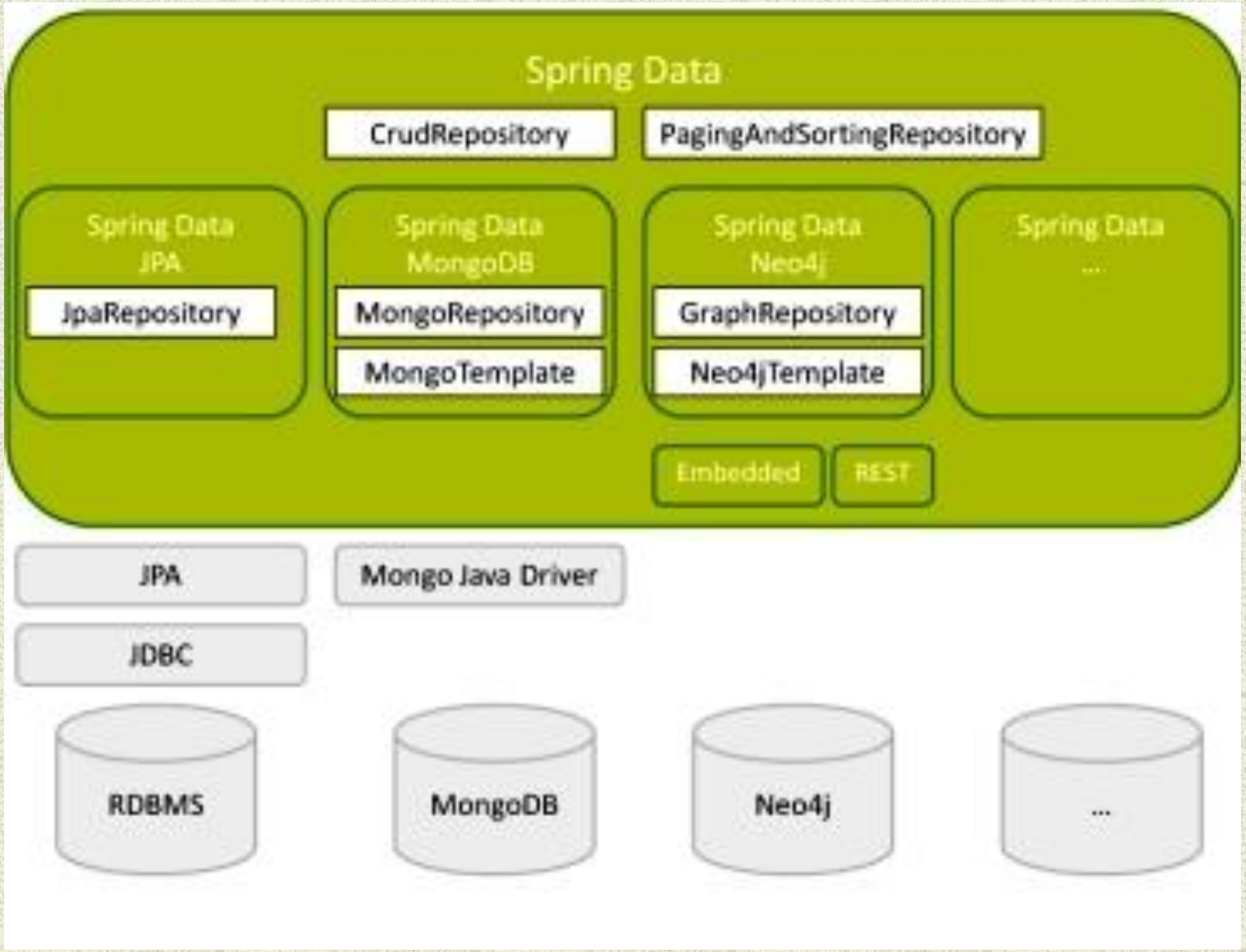
Hibernate ORM

(Hibernate for short) is an object-relational mapping Java library;

a framework for mapping an object-oriented domain model to a

traditional relational database. Distributed under the GNU Lesser General Public License

Spring Data Project



Spring Data Repositories

Spring Data repository abstraction

Significantly reduce the amount of boilerplate code required to implement data access layers

Domain Object specific wrapper that provides capabilities on top of EntityManager

Performs function of a Base Class DAO

Three Types:

CrudRepository provides CRUD functions .

•

PagingAndSortingRepository provide methods to do pagination and sorting records.

JpaRepository provides methods such as flushing the persistence context and delete record in a batch

JPA ORM Fundamentals

- EntityManager
  - API is used to access a database
  - Basically a CRUD Service PLUS { persist, find, remove}.
  - Can Find entities by their primary key, and to query over all entities.
  - Can participate in a transaction.
- Transaction Manager
  - Java Transaction API
  - General API for managing transactions in Java
  - Start, Close, Commit, Rollback operations
- Entity
  - lightweight persistence domain object
  - Annotation driven Entities - @Entity

Wiring the Components

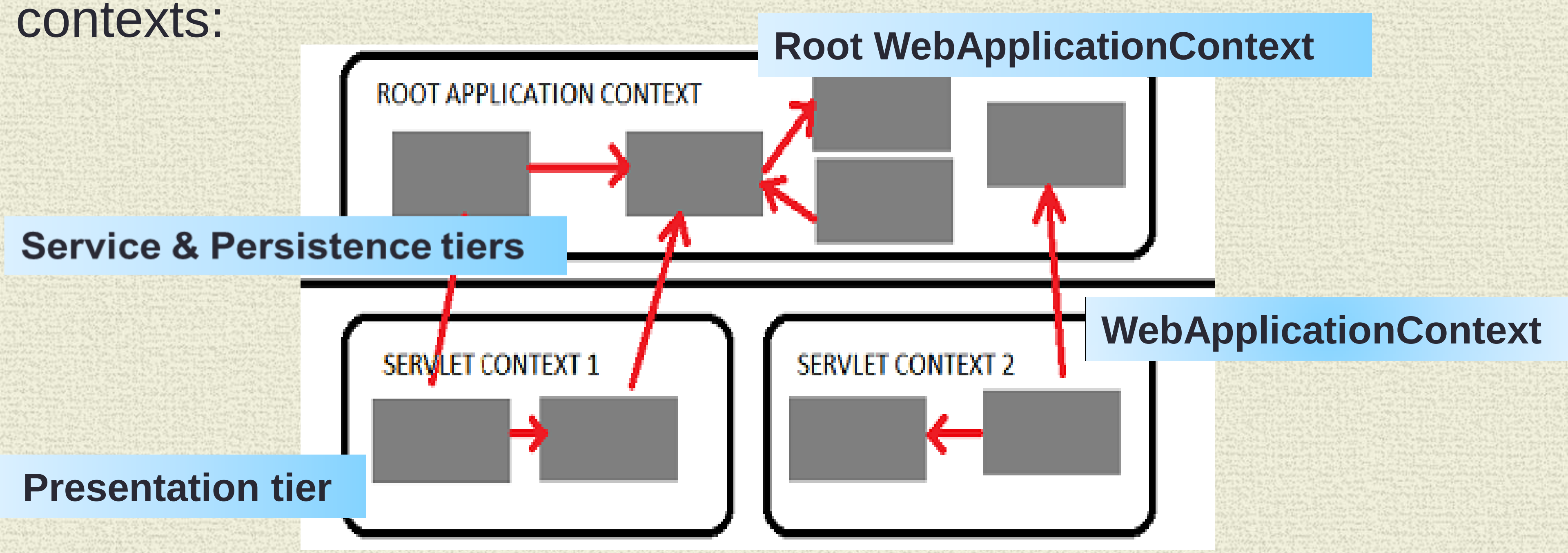
ApplicationContext.XML

```
 <bean id="entityManager"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
       <property name="packagesToScan"
value="com.packt.webstore.persistence.domain" />

<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManager" />

 <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.h2.Driver"/>
```

Web Application Context

Spring has multilevel application context hierarchies.

Web apps by default have two hierarchy levels, root and servlet contexts:



Presentation tier has a WebApplicationContext [Servlet Context] which inherits all the resources already defined in the root WebApplicationContext [Services, Persistence]

Wiring Continued

Scan for interfaces extending Spring Data Repository

```
<jpa:repositories base-package="com.packt.webstore.domain.repository"/>
```

Scan for components in service/persistence tiers

```
<context:component-scan base-package= "com.packt.webstore.service" />
<context:component-scan base-package= "com.packt.webstore.repository"/>
```

Enable transactions for transaction-based resources

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

Implementation Details

```java
@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {
  Product getProductById(Long id);
  Product getProductByProductId(String productId);
```

```java
@Service
@Transactional
public class OrderServiceImpl implements OrderService{
@Autowired
private ProductRepository productRepository;
  public void processOrder(String productId, long quantity) {
Product producId = productRepository.getProductByProductId(productId);
```

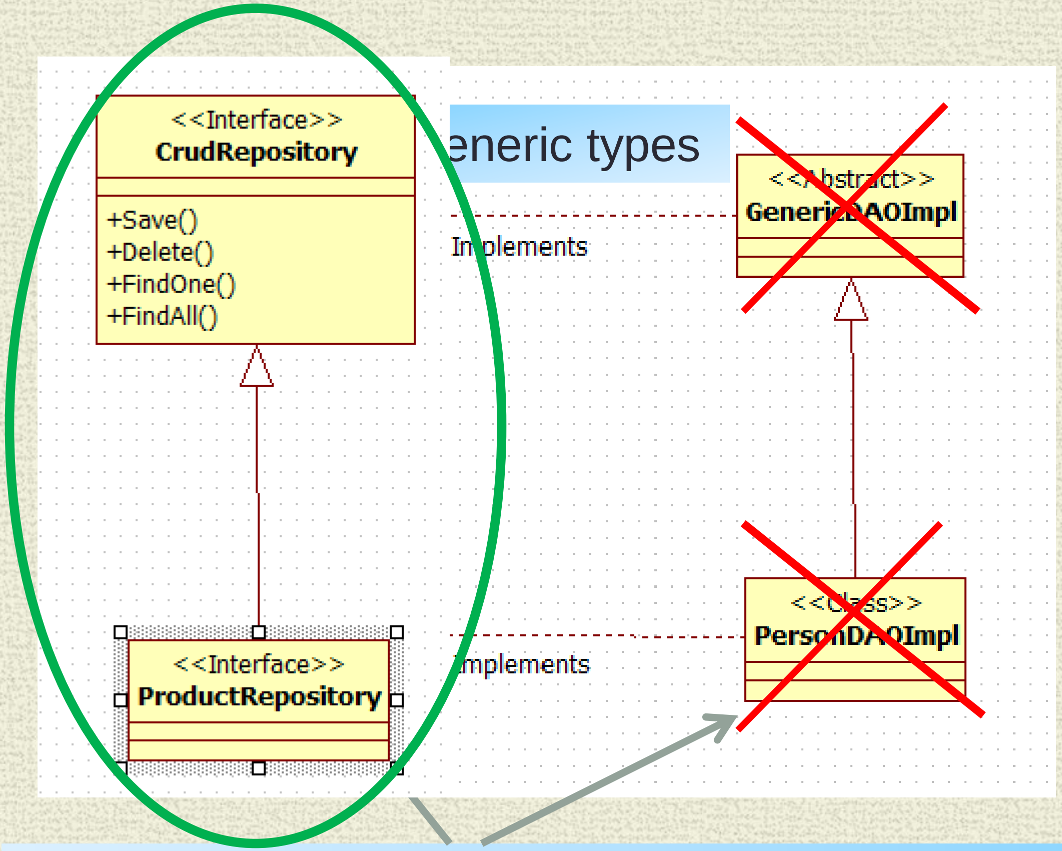
```java
@Entity(name = "PRODUCT")
public class Product {
@Id
private Long id;
@Column(name = "PRODUCTID")
private String productId;
```

CrudRepository

- public **interface** CrudRepository<T, ID extends Serializable> **extends Repository**<T, ID> {
-     <S extends T> S save(S entity);
-     T findOne(ID primaryKey);
-     Iterable<T> findAll();
-     Long count();
-     void delete(T entity);
-     boolean exists(ID primaryKey);
- }
- LOOKS just Like [what is Known as] a “generic DAO interface”
- HOWEVER, Spring provides [default] implementations – effectively Java 8-like default methods in an interface

"Classic" ORM GenericDAO

Spring Data



Adds Domain Object specific functionality

Spring Version of DAO

```
@Service
@Transactional
public class ProductServiceImpl implements ProductService {

@Autowired
ProductRepository productRepository;      // "ProductDAOImpl"

public void createProduct(Product product)  {
        productRepository.save(product );
}
```

ProductDAOImpl

```
@Repository
public interface ProductRepository extends  CrudRepository<Product, Long> {  }
```

"Automatically" creates CRUD services

Main Point

- JPA is a specification not an implementation. It provides a consistent, reliable mechanism for data storage and retrieval.

- *TM is a reliable mechanism of transcending for consistent access to the source of thought.*

Persistence Context ~= Hibernate Session

ORM Session

Spring “manages” through @Transactional

- Unit of work

  Common Pattern: *session-per-request*

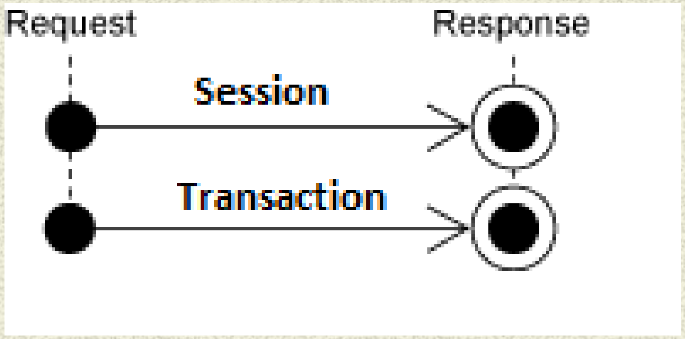
  Session == Database Transaction

- **START –**

  Open a Persistence Context

  Open a single database connection

  Start a Transaction



- **Do the Work –**

  Associate & Manage entities W/R the session

  Exercise DB CRUD operations

- **END –**

  End Transaction

  Close a Persistence Context

.

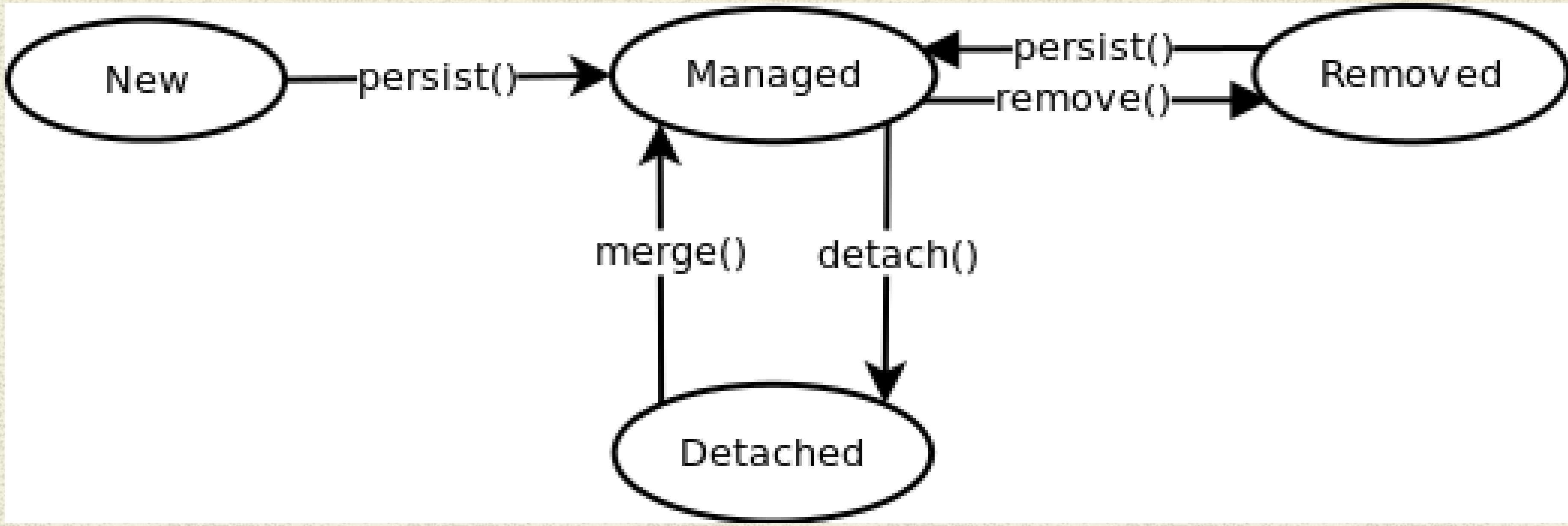
An Entity's Object States Relationship with the ORM Session

- ***Transient –***
- it has just been instantiated using the new operator
- not associated with a Persistence Context
- no persistent representation in the database
- ***Persistent –***
- representation in the database
- Has been saved or loaded in Persistence Context

- Changes made to an object are synchronized with the database when the unit of work completes..
- ***Detached –***
- An object that has been persistent - its Session has been closed

ORM Entity Lifecycle

- **Transient** **Persistent** **Removed**



Detached

Configurable Parent-Child operations Thru CASCADE TYPES

Helps Manage the state of complex objects

- **Persist**
- If the parent is persisted so are the children
- **Remove**
- If the parent is "removed" so are the children
- **Merge [a detatched object]**
- If the parent is merged so are the children
  - Merge modifications made to the detached object are merged into a corresponding **DIFFERENT** managed object

Configurable Parent-Child operations [Some] Fetching Strategies

- *Immediate fetching*: an association, collection or attribute is fetched immediately when the owner is loaded. **[(JPA)Default for one-to-one]**

- *Lazy collection fetching*: a collection is fetched when the application invokes an operation upon that collection. **[Default for collections]**

Configurable Parent-Child operations Demo - Fetch - Cascade Example

- **public class Customer{**
- `@OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)`
- `@JoinColumn(name="customerId")`
- `private List<Product> productList ;`
- }

- `FetchType.LAZY` means collection is NOT fetched until collection element is referenced…

It also means that you will get a LazyInitializationException

If you try to reference it after the PersistenceContext is closed!!

- ***`CascadeType.ALL`*** means collection is persisted, merged or refreshed when parent is.

ORM Parent-Child “Relationships”

One-to-One

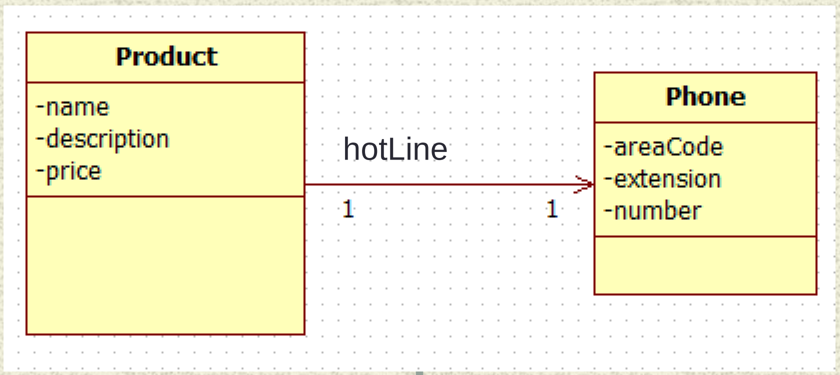
One-to-Many

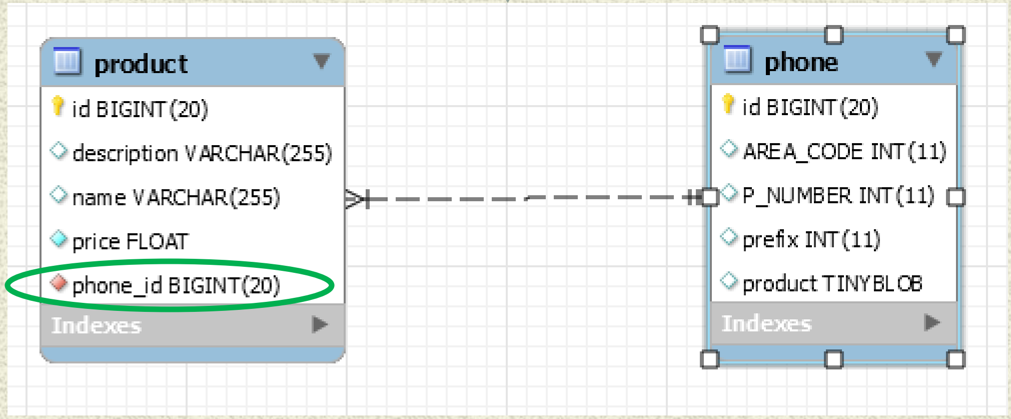
Many-to-Many

Unidirectional – Bidirectional

See Demo ProductJPA

OneToOne Unidirectional

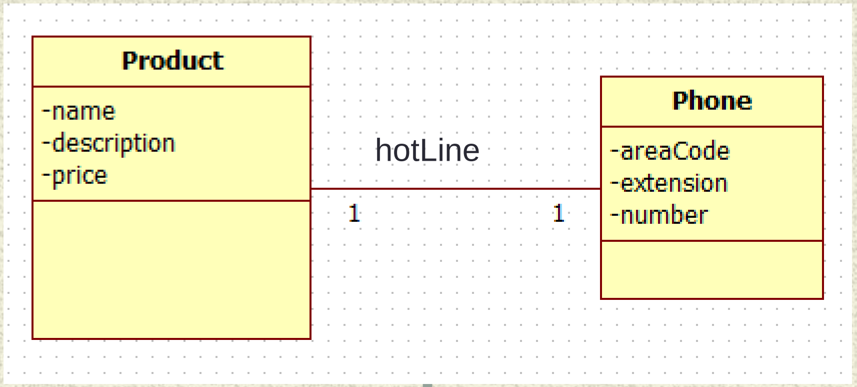


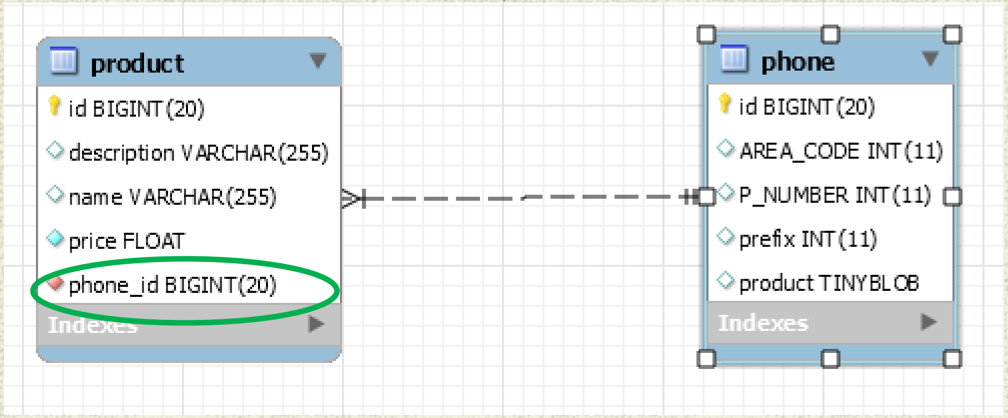


OneToOne Unidirectional

```
• @Entity public class Product{
        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        private long id; ...

•       @OneToOne(cascade = CascadeType.ALL)
•       @JoinColumn(name="phone_id", unique = true )
        private Phone hotLine; ...
• }
```

OneToOne Bi-directional





OneToOne Bi-directional

Annotation the OTHER side of the relationship ALSO…

```
@Entity
 public class Phone{
        @Id
        private long id; ...
       @OneToOne(mappedBy="hotLine",
                        cascade=CascadeType.ALL)
        private Product product; ...
}
```

mappedBy – use the foreign key and mapping in the *source* to define the *target* mapping

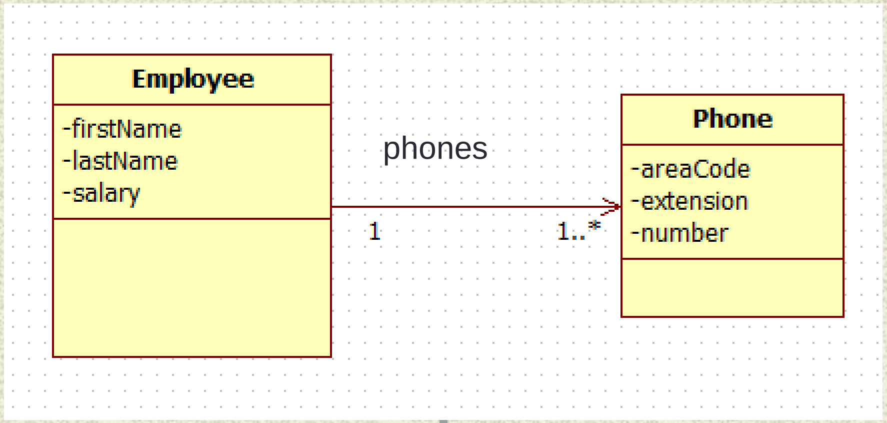
Bi-directional Relationships

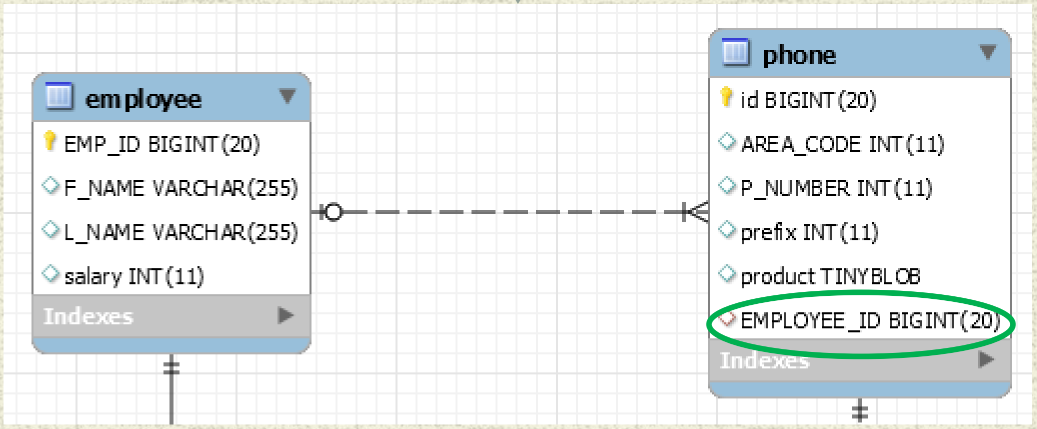
WARNING NOTICE

- If you add or remove to one side of the collection, you **must** also add or remove from the other side
- Database will be updated correctly **ONLY** if you add/remove from the owning side of the relationship
- Your object model can get out of synch if you do not pay attention…

OneToMany Unidirectional

•



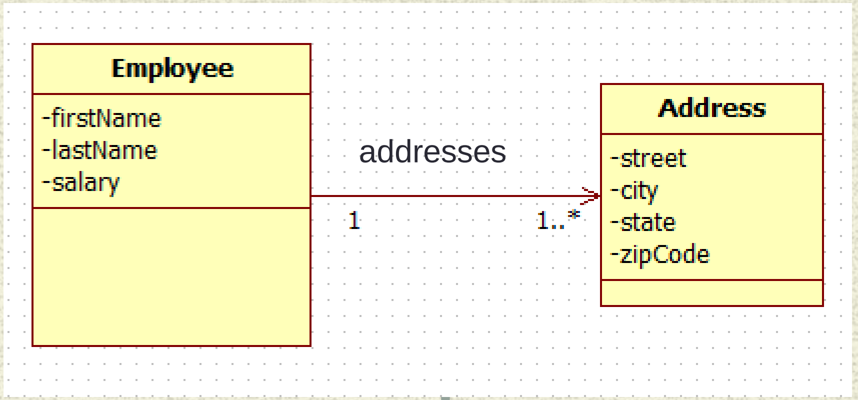


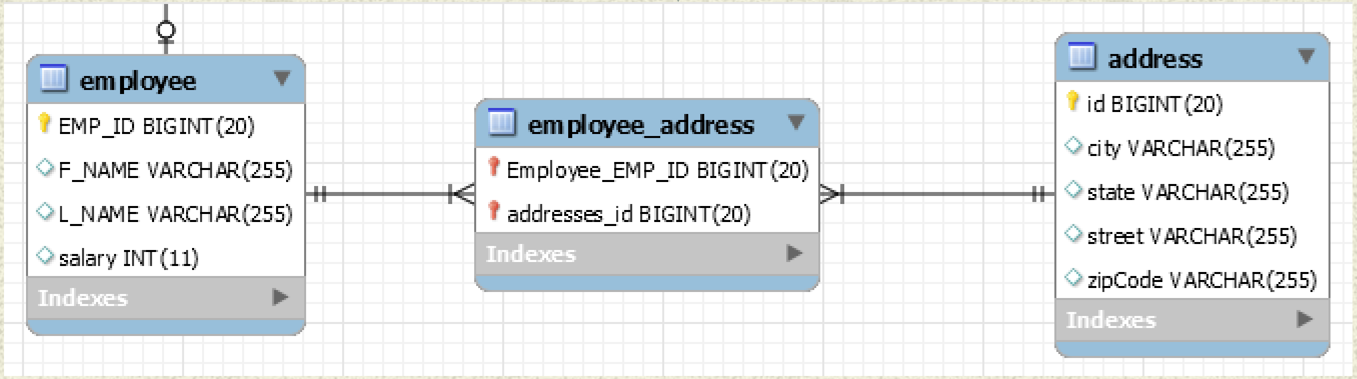
OneToMany Unidirectional JoinColumn

```
@Entity
public class Employee {
      @Id
      @Column(name="EMP_ID")
       private long id; ...

      @OneToMany
      @JoinColumn
      private List<Phone> phones; ...
}
```

HIBERNATE REFERENCE DOC:
A *unidirectional one-to-many association on a foreign key* is an unusual case, and is not recommended. You should instead use a join table for this kind of association.

One-to-Many Join Table





OneToMany Unidirectional JoinTable

```
@Entity public class Employee {
  @Id @Column(name="EMP_ID")
  private long id; ...

                     This is the Default
@OneToMany
private      Set<Address> addresses;
}
```

OneToMany Bi-directional JoinColumn

```
@Entity
public class Employee {
   @Id
   @Column(name="EMP_ID")
   private long id; ...

  @OneToMany(mappedby ="employee")
  private List<Address> addresses; ...
}
```

```
@Entity
public class Address{
   @Id
   private long id; ...

   @ManyToOne
   @JoinColumn(name="EMP_ID")
   private Employee employee;
}
```

Owns relationship

NOTE: JoinColumn OPTIONAL
Bidirectional DEFAULTS to Join Column

OneToMany Bidirectional JoinTable

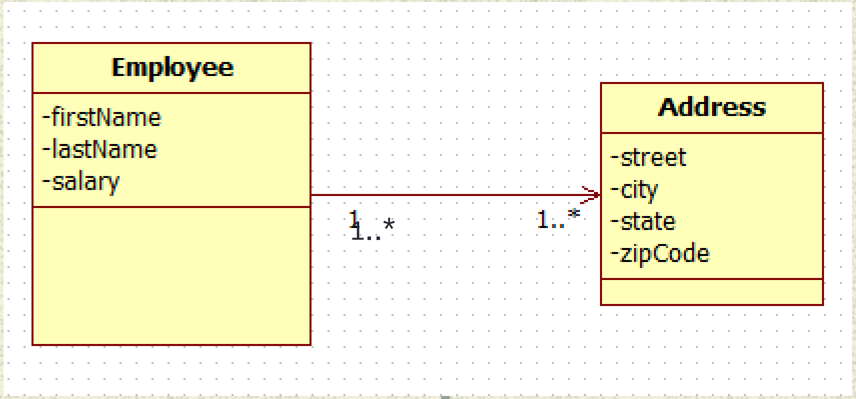
OneToMany side same as unidirectional example

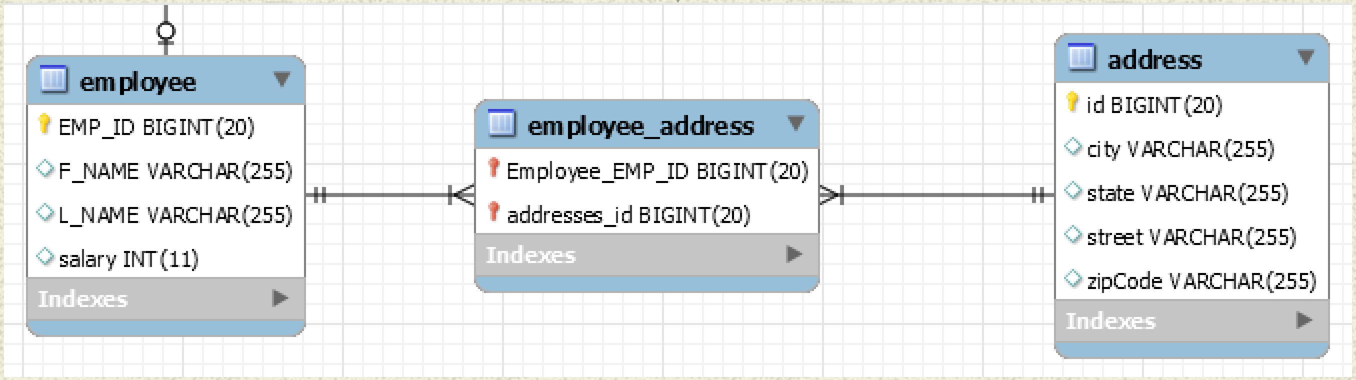
```
@Entity
public class Employee {
   @Id
   @Column(name="EMP_ID")
   private long id;
 @OneToMany
    private      Set<Address> addresses;
}
```

Simply Add ManyToOne on child object

```
@Entity
public class Address {
…
@ManyToOne
@JoinTable ( name="Address_Member")
private Employee employee;
```

Many-to-Many


Employee
-firstName
-lastName
-salary
1..*
1..*
Address
-street
-city
-state
-zipCode


employee
EMP_ID BIGINT(20)
F_NAME VARCHAR(255)
L_NAME VARCHAR(255)
salary INT(11)
Indexes
employee_address
Employee_EMP_ID BIGINT(20)
addresses_id BIGINT(20)
Indexes
address
id BIGINT(20)
city VARCHAR(255)
state VARCHAR(255)
street VARCHAR(255)
zipCode VARCHAR(255)
Indexes

Many-To-Many

```
@Entity
public class Employee {
   @Id
   @Column(name="EMP_ID")
   private long id;

@ManyToMany
   private      Set<Address> addresses;
```

- If Converting from OneToMany [Join table] –
- The ManyToMany is achieved by simply dropping the unique constraint on the JoinTable created by OneToMany

JPQL - Data Object Queries

JPA Query Language

- JPQL is similar to SQL, but operates on objects, attributes and relationships instead of tables and columns.

```
public class Product implements Serializable {
 private String name;
 @OneToOne
 private Phone hotLine;
```

- `JPQL:`

  SELECT p FROM Product p
- Will Yield:

  Product with Phone;
- Where:

  product.getHotLine().getNumber(); is populated

NOTE: JPA `OneToOne relationship defaults to eager`

Spring Data Repository Query Resolution Query examples

- **CREATE example**
- `Employee findById(long id);`

USE DECLARED QUERY example

- `@Query("SELECT e FROM Employee e where EMP_ID = :id")`
- `Employee findById(@Param("id") long id);`

- **Use class level declared query EXAMPLE**
-
```
public static final String FIND_BY_ID_QUERY = "select e from
           Employee e where EMP_ID = :id";
```

- `@Query(FIND_BY_ID_QUERY)`
- `Employee findById(@Param("id") long id);`

Main Point

- An ORM framework provides capabilities [fetch, cascade, relationship mapping] that facilitates conversion between OO and relational data representations

- *Studies show that increased brain wave coherence & TM facilitate the learning of new concepts.*