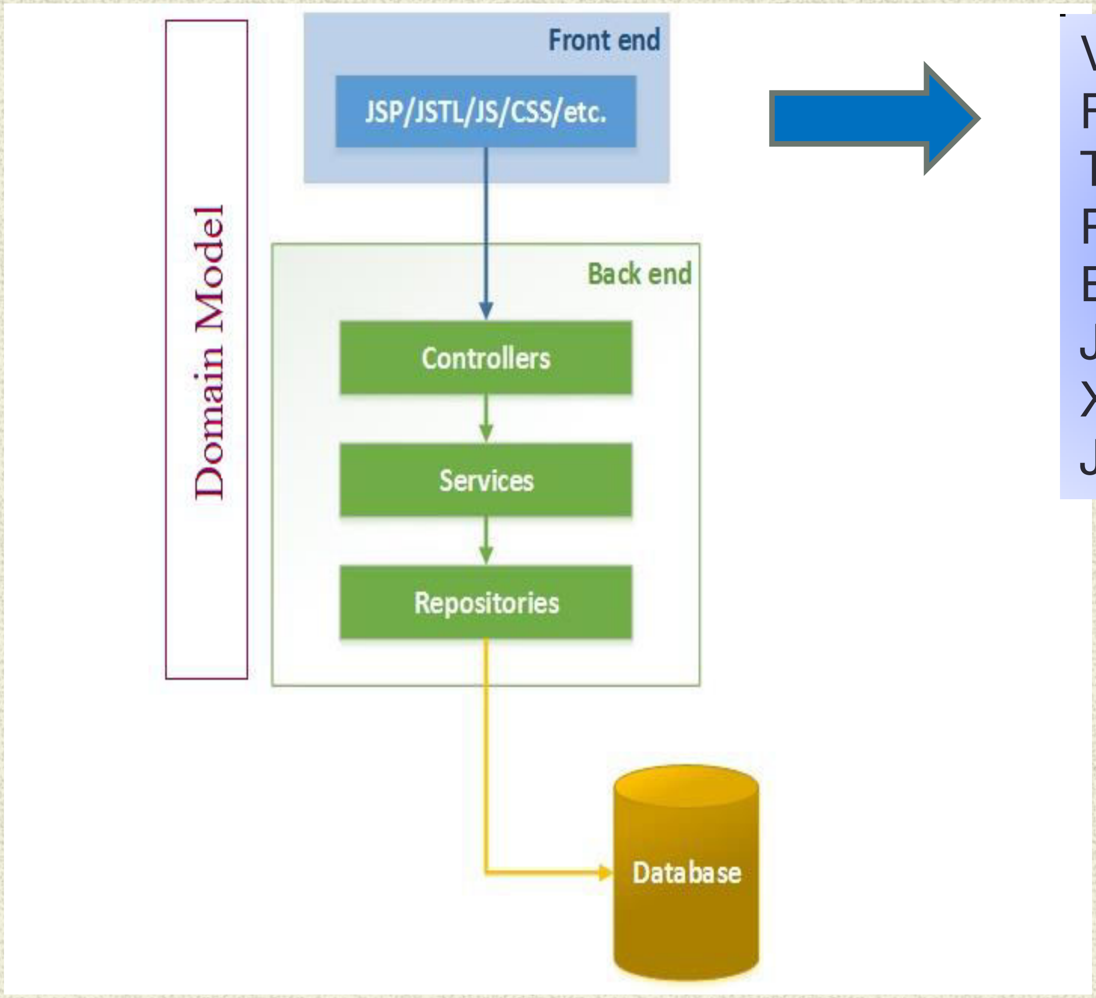
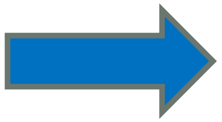
LESSON 10
PERSISTENCE & TRANSACTIONS
TAPPING THE SOURCE OF PURE KNOWLEDGE

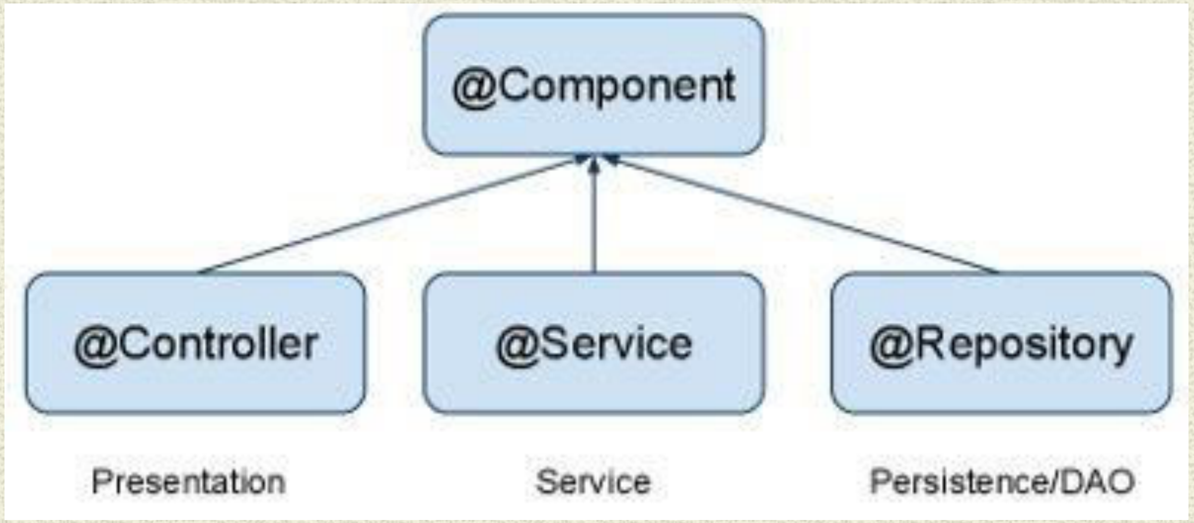
Spring MVC Layers





Velocity
FileMarker
ThymeLeaf
PDF
Excel
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]

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

Main Point

- An N Tier Architecture separates an application into layers thereby supporting a separation of concerns making any application more efficient, modular and scalable. *Life is structured in layers. It is a structure that is both stable and flexible, consistent yet variable and it encompasses an infinite range of possibilities*

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**

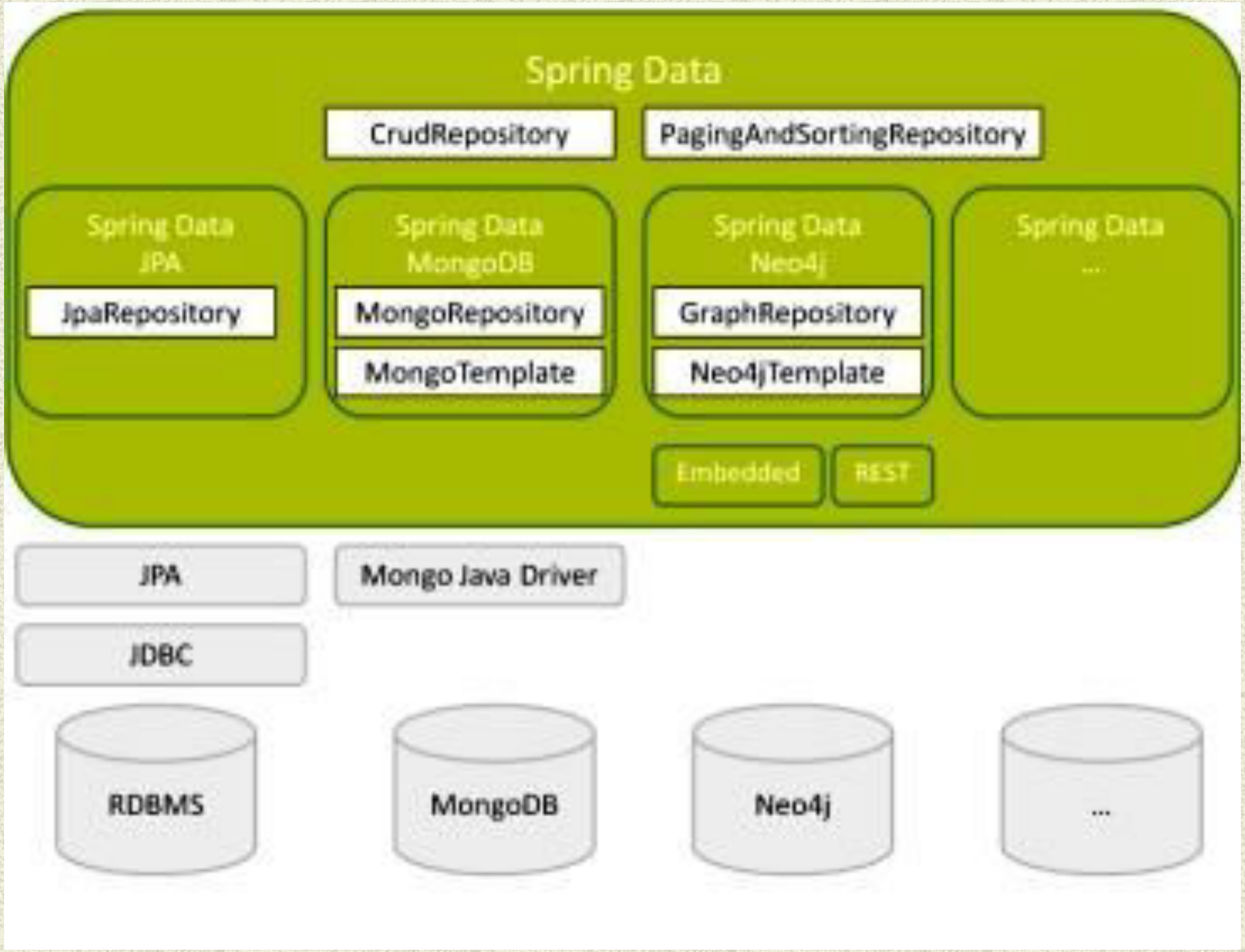
- Major Implementations [since 2006]:
  - Toplink - Oracle implementation [donated to Eclipse foundation for merge with Eclipselink 2008]
  - Hibernate - Most deployed framework. Major contributor to JPA specification.
  - OpenJPA - (openjpa.apache.org) which is an extension of Kodo implementation.

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



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.
- JTA
  - Java Transaction API
  - General API for managing transactions in Java
  - Start, Close, Commit, Rollback operations
  - Handles distributed transactions
- Entity
  - lightweight persistence domain object
  - Annotation driven Entities - @Entity

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

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 .

ORM caching mechanisms

Persistence provider manages local store of entity data

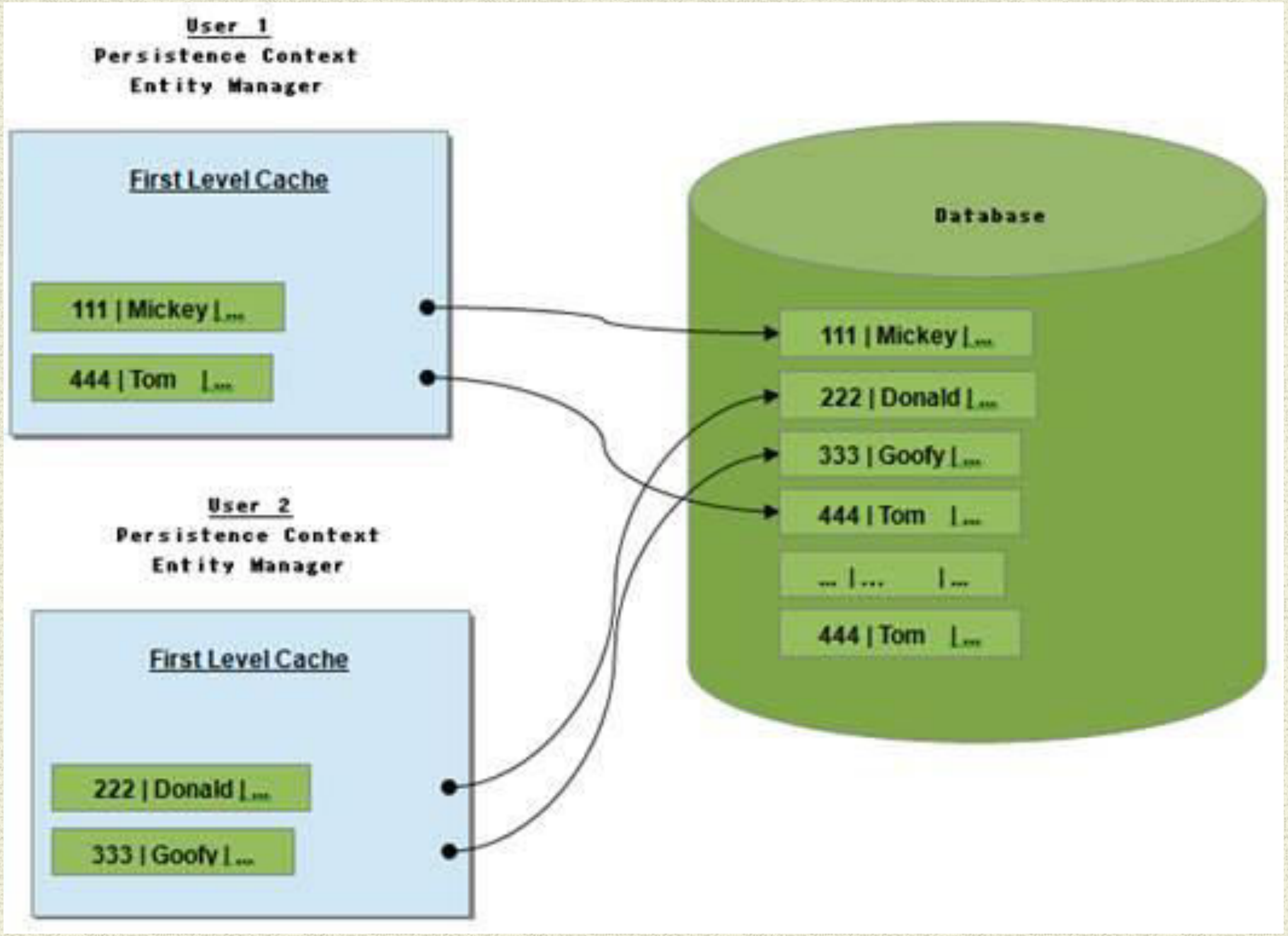
- Leverages performance by avoiding expensive database calls
- Data are kept transparent to the application
- CRUD operation can be performed through normal entity manager functions
- Application can remain oblivious of the underlying cache and do its job without concern

- Level 1 Cache

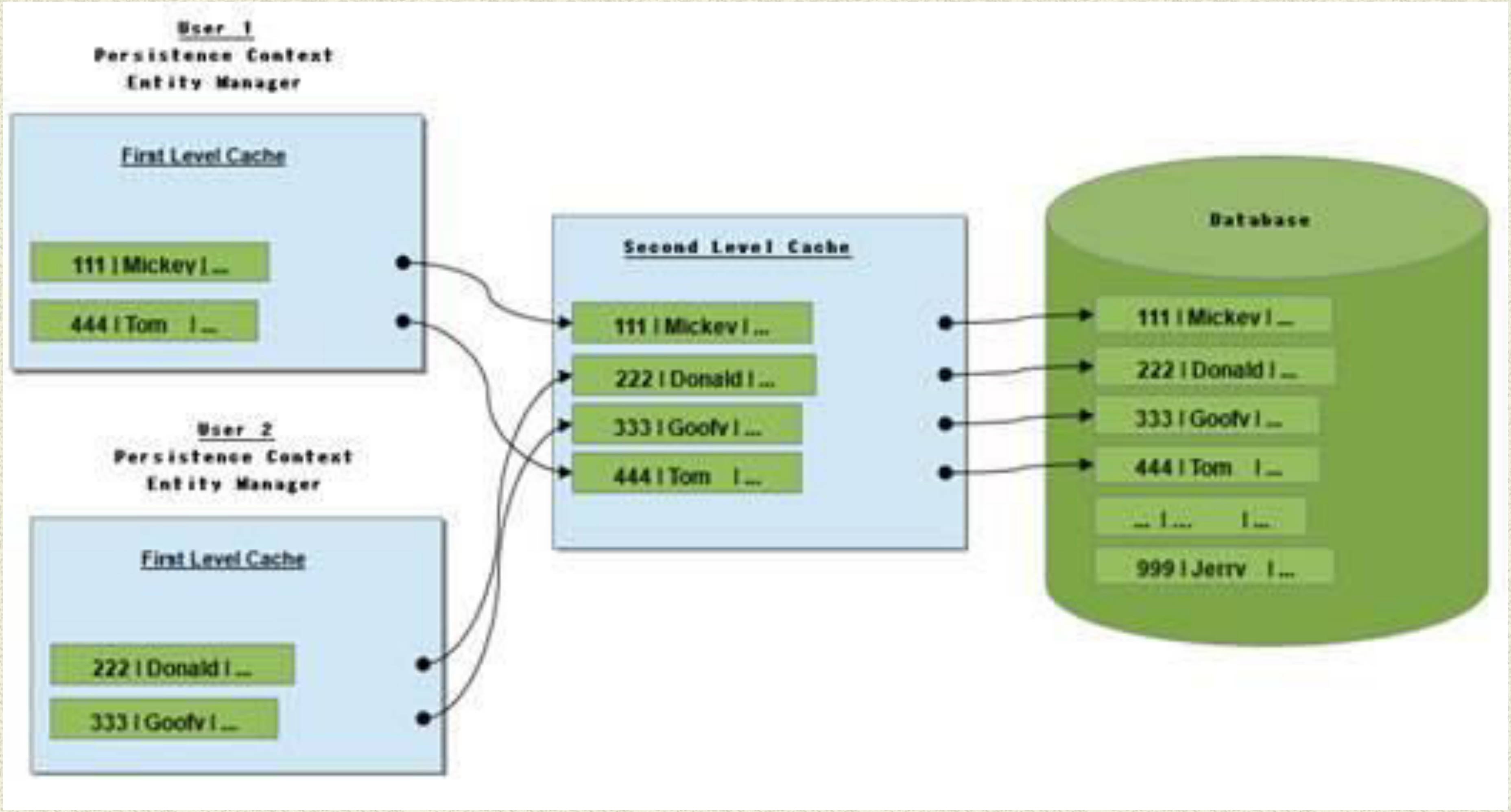
  Available within the same transaction [session]
- Level 2 Cache

  Available throughout the application.

Level 1 Cache

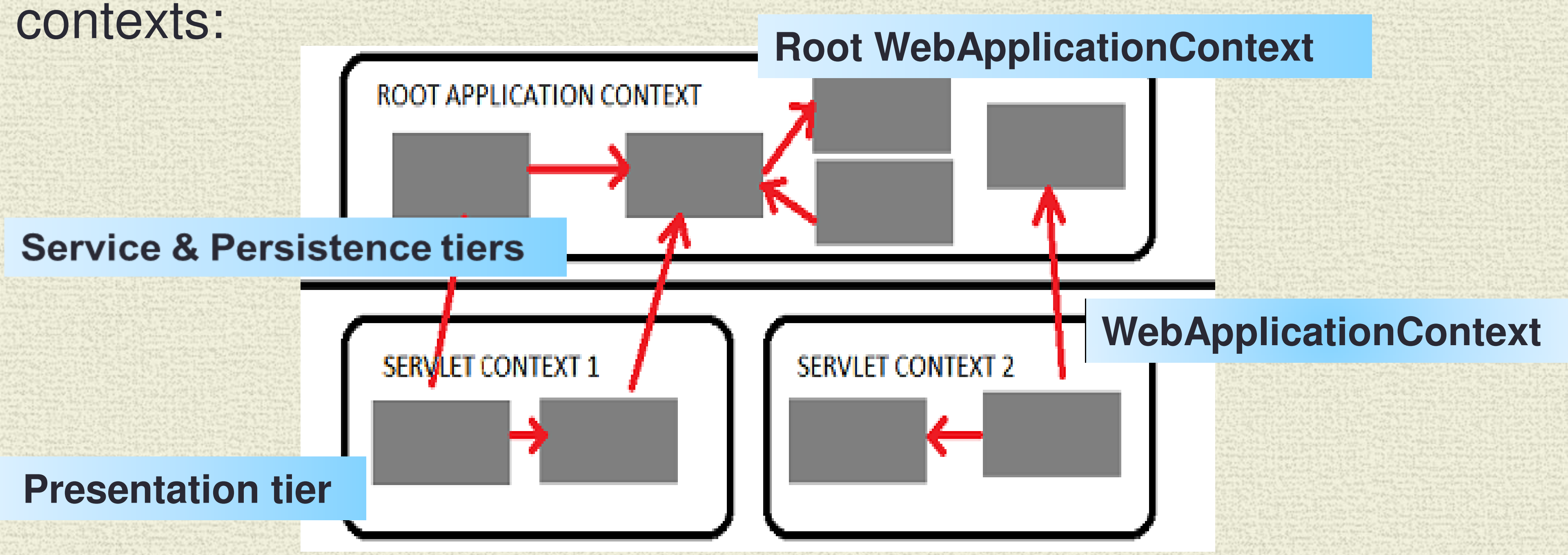


Level 2 Cache



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


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

Wiring Continued

Scan for interfaces extending Repository

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

Scan for transaction-based resources

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

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

Implementation Details

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

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

- @Service
- public class OrderServiceImpl implements OrderService{
- @Autowired
- private ProductRepository productRepository;

  public void processOrder(String productId, long quantity) {
- Product producId = productRepository.getProductByProductId(productId);

JPA EntityManagers

Container (e.g.,Spring) manages the lifecycle of the Entity Manager,

Container Managed & Transaction Scoped Entity Managers

Transaction exists? – join it

No Transaction? Start one

Transaction ends when leaving the method that first accesses the EM

Container Managed & Extended Scope Entity Managers

Read only operations can be executed outside a transaction

Modifications executed outside a transaction queued until a transaction is joined

Application Managed Entity Managers

Explicit control - how and when EntityManager should be used.

- The EntityManager invokes the Hibernate session under the hood. You can obtain the session by calling:
- Session session = entityManager.unwrap(Session.class);

- The CrudRepository invokes the JPA EntityManager under the hood. You can access the EntityManager by injecting it:
- @Autowired
- EntityManager entityManager;
-

Session & Transaction Management

CONCEPTUAL DIFFERENCES:

Hibernate managed
Application managed EntityManager
Container managed EntityManager
Spring Managed

Generic DAO interface example DataSource neutral

```
public interface GenericDao<T> {
    long countAll(Map<String, Object> params);
    T create(T t);
    void delete(Object id);
    T find(Object id);
    T update(T t);
}
```

- public class GenericDaoHibernateImpl<T, PK> extends Serializable> implements GenericDao<T, PK> {

- @Autowired
-    SessionFactory sessionFactory;

-    public void create(T entity){
-       getCurrentSession().persist(entity);
-    }

JPA Container Managed Implementation Generic DAO Interface

- public class GenericDaoJpaImpl<T, PK> extends Serializable> implements GenericDao<T, PK> {

- @PersistenceContext
- EntityManager em;

- public T create(T t) {

  this.em.persist(t);

  return t;

  }

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);
- }

Hibernate (DAO)

HibernateUtil “manages“ Session Factory & Session singleton
ALSO can manage transactions….SEE example code

```
Session session=
        HibernateUtil.getSessionFactory(). getCurrentSession();

public void save(Employee employee  ) {

   Transaction t= session.beginTransaction();
   try {
        session.persist(employee );

        t.commit();
     } catch(Exception ex) {
        t.rollback();
        throw ex;
     }
  session.close();
```

Application Managed JPA Entity Manager(DAO)

```
EntityManager  entityManager =
                              HibernateJpaUtil.getEntityManager();
```

HibernateJpaUtil “manages” EntityManager Factory & EntityManager Singleton

```
public void save(Employee employee) {
          UserTransaction t = entityManager.getTransaction();
          try {
                    t.begin();
                     entityManager.persist(employee );
                    t.commit();
          } catch(Exception ex) {
                    t.rollback();
                    throw ex;
           }
          entityManager.close();
```

Container Managed and Transaction Scoped Entity Manager(DAO)

```
@PersistenceContext
protected EntityManager entityManager;

public void save(Employee employee) {
          entityManager.persist(employee );
   }
```

Spring Version

```
@Service
@Transactional
public class EmployeeServiceImpl implements EmployeeService {

@Autowired
EmployeeRepository employeeRepository;      // “DAO”

public void createEmployee(Employee employee)  {
          employeeRepository.save(employee );
}
```

DAO

- @Repository
- public interface EmployeeRepository extends CrudRepository<Employee, Long>
- {   }

Entity - Object States

- ***Transient –***
- it has just been instantiated using the new operator
- not associated with a Session
- no persistent representation in the database
- ***Persistent –***
- representation in the database
- Has been saved or loaded in Session
- Changes made to an object are synchronized with the database when the unit of work completes..
- ***Detached –***
- An object that has been persistent, but its Session has been closed

Configurable Parent-Child operations CASCADE TYPES Manage state of object

- **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
- **Refresh**
- If the parent is refreshed so are the children
  - Refresh the state of the instance from the database, overwriting changes made to the entity

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]**

- *"Extra-lazy" collection fetching*: individual elements of the collection are accessed from the database as needed. Hibernate tries not to fetch the whole collection into memory unless absolutely needed. It is suitable for large 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 = new ArrayList<Product>();`
- }

- `FetchType.LAZY` means collection is NOT fetched until collection element is referenced
- `CascadeType.ALL` means collection is persisted, merged or refreshed when parent is.

ORM Parent-Child “Relationships”

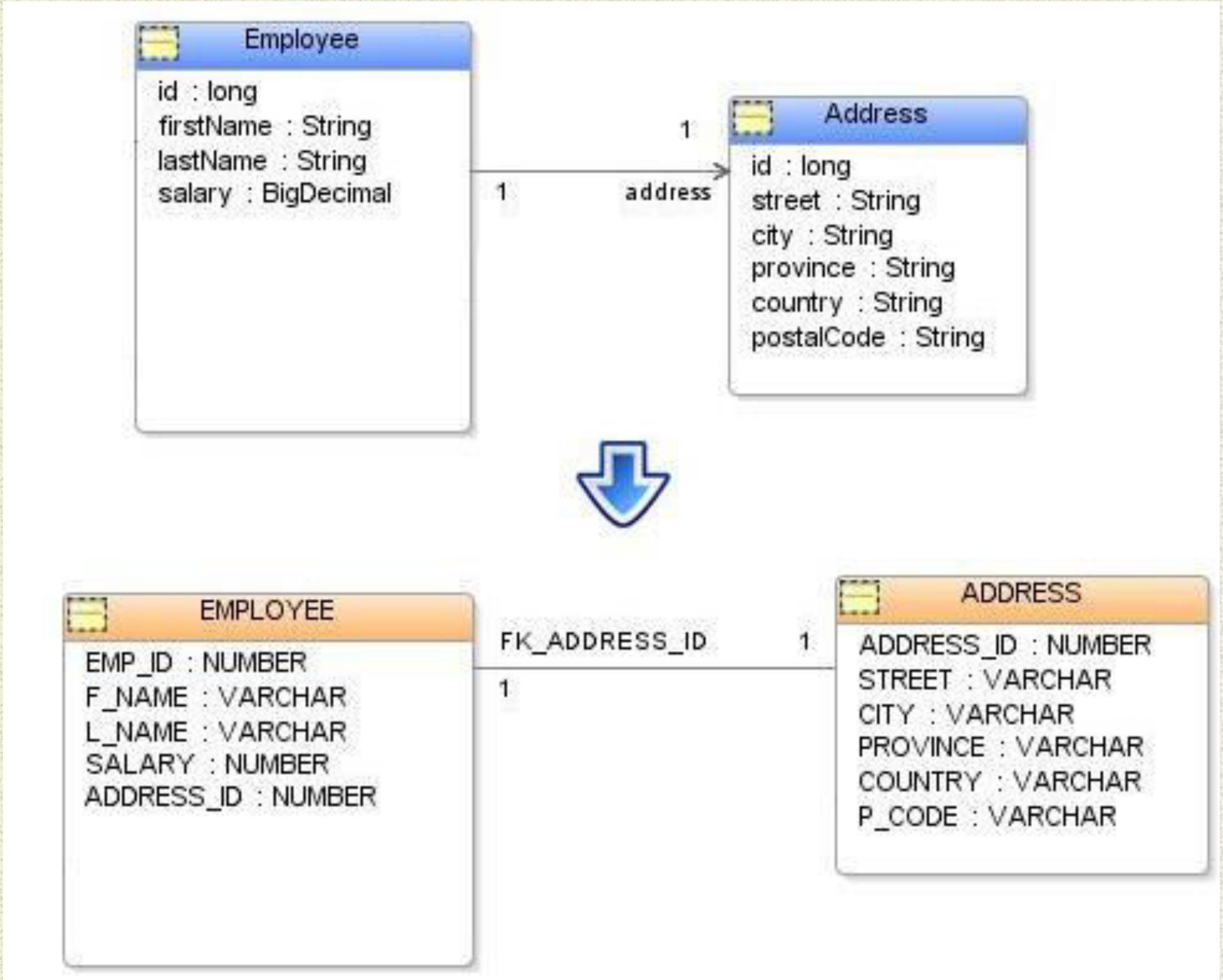
One-to-One

One-to-Many

Many-to-Many

Unidirectional - Bidirectional

OneToOne Unidirectional



OneToOne Unidirectional

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

        @OneToOne
        @JoinColumn(name="address_id")
        private Address address; ...
• }
```

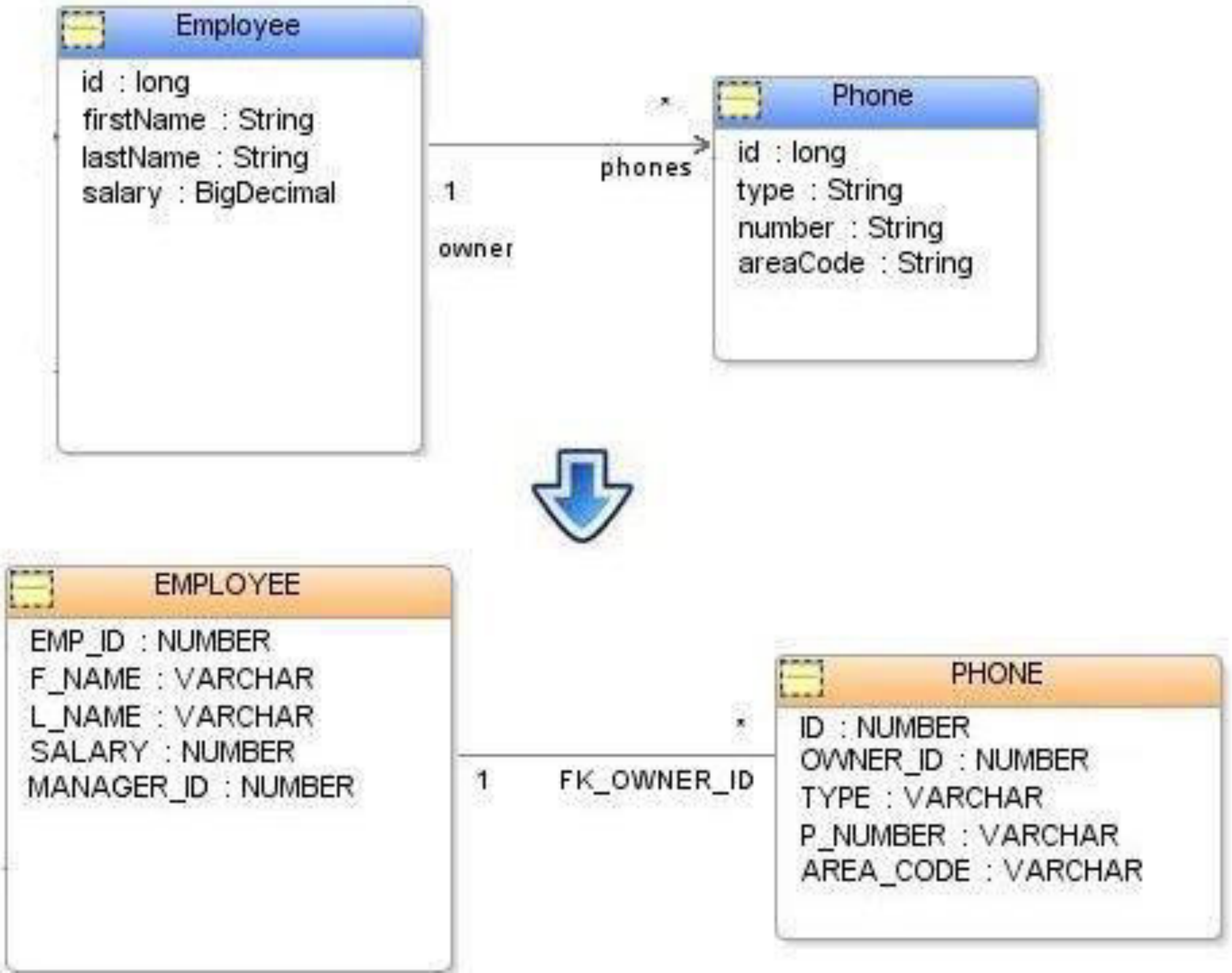
OneToOne Bi-directional

Annotation the OTHER side of the relationship ALSO…

```
@Entity
 public class Address {
         @Id
         private long id; ...
         @OneToOne(fetch=FetchType.LAZY,mappedBy="address")
         private Employee owner; ...
}
```

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

OneToMany Unidirectional



OneToMany Unidirectional JoinColumn

Your mapping declares that the many side of the relationship knows nothing about its parent. However in database terms that entity has the foreign key to the one side and does know about its parent. Therefore your mapping and database structure do not agree. It is a “confusing: design.

Having a join table eliminates that mismatch.

Also multiple parents to [same] child   will complicate the child end. It requires a foreign key per parent type. For example a Task can be the child of Project, Person, Department, Study, and Process.

With a one-to-many JoinColumn**…the sequence is:**

insert the child record,
insert the parent,
update the child with the parent's key…

association.

OneToMany Unidirectional JoinTable

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

@OneToMany
 @JoinTable ( name="EMP_ADDRESS",
   joinColumns={ @JoinColumn(name="EMP_ID"},
  inverseJoinColumns={ @JoinColumn(name="ADDRESS_ID", unique=true) }
)
   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<Phone> phones; ...
}
```

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

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

Owns relationship

You can also map a bidirectional one to many, with the one-to-many side as the owning side, you have to remove the mappedBy element and set the many to one @JoinColumn as insertable and updatable to false. This solution is not optimized and will produce some additional UPDATE statements. – Hibernate Reference

OneToMany Bidirectional JoinTable

OneToMany side same as unidirectional example

```
@Entity
public class Employee {
   @Id
   @Column(name="EMP_ID")
   private long id;
 @OneToMany
 @JoinTable ( name="EMP_ADDRESS",
    joinColumns={ @JoinColumn(name="EMP_ID"},
   inverseJoinColumns=
        {@JoinColumn(name="ADDRESS_ID", unique=true) } )
    private       Set<Address> addresses;
}
```

Simply Add ManyToOne on child object

```
@Entity
public class Address {
…
@ManyToOne
private Employee employee;
```

Many-To-Many

- Drop the unique Constraint on the JoinTable OneToMany

Changed to ManyToMany

```
@ManyToMany
 @JoinTable ( name="EMP_ADDRESS",
   joinColumns={ @JoinColumn(name="EMP_ID"},
  inverseJoinColumns={
@JoinColumn(name="ADDRESS_ID ) } )
   private       Set<Address> addresses;
```

Removed - unique=**true**

Spring Data Repository Query Resolution

- **CREATE**
- CREATE attempts to construct query from the query method name.
- **USE_DECLARED_QUERY**
- USE_DECLARED_QUERY tries to find a declared query and will throw an exception in case it can't find one. The query can be defined by an annotation somewhere or declared by other means.
- **CREATE_IF_NOT_FOUND (default)**
- CREATE_IF_NOT_FOUND combines CREATE and USE_DECLARED_QUERY. It looks up a declared query first, and if no declared query is found, it creates a custom method name-based query.

Query examples

- `<jpa:repositories base-package="jpa"`
  `query-lookup-strategy="create-if-not-found" />`

- **CREATE - CREATE_IF_NOT_FOUND example**
- `Product getProductByProductId(String key);`

USE_DECLARED_QUERY example

- `@Query("SELECT p FROM PRODUCT p where PRODUCTID = :productId")`
- `Product getProductByProductId(@Param("productId") String key);`

- **JOIN EXAMPLE**
- public final static String FIND_BY_ADDRESS_QUERY = "SELECT p " + "FROM Person p LEFT JOIN p.addresses a " + "WHERE a.address = :address";
- @Query(FIND_BY_ADDRESS_QUERY)
- public List<Person> findByAddress(@Param("address") String address)

JPA Named Query

- Declaration:
- *@Entity*
- *@NamedQuery(name = "User.findByEmailAddress", query = "select u from User u where u.emailAddress = :emailAddress")*
- **public** class User { }

- Usage:
- **public** interface UserRepository extends CrudRepository<User, Long> {

```
        List<User> findByLastname(String lastname);
        User findByEmailAddress(String emailAddress);
}
```

Main Point

- JPA is a specification not an implementation. It provides a consistent, reliable mechanism for data storage and retrieval that alleviates the application developer from the details involved in the persistence layer. *The mechanism of transcending allows the individual to tap into Transcendental Consciousness and enlivens its qualities in activity.*

- A web application can define any number of DispatcherServlets. Each servlet will operate in its own namespace, loading its own application context with mappings, handlers, etc. Only the root application context as loaded by ContextLoaderListener, if any, will be shared.

One-to-Many [Join Table; One-to-One Join Column,

- ALTER TABLE PRODUCT ADD COLUMN customerId BIGINT;

- ALTER TABLE PRODUCT ADD CONSTRAINT CUSTOMERREL FOREIGN KEY (customerId) REFERENCES CUSTOMER (id);