



# Spring and Hibernate

*Nilesh Ghule <nilesh@sunbeaminfo.com>*



# Hibernate



# Hibernate 5 Bootstrapping

```
public class HbUtil {  
    private static final SessionFactory factory  
        = createSessionFactory();  
    private static ServiceRegistry serviceRegistry;  
  
    private static SessionFactory createSessionFactory() {  
        ① serviceRegistry = new StandardServiceRegistryBuilder()  
            .configure() // read from hibernate.cfg.xml  
            .build();  
  
        ② Metadata metadata = new MetadataSources(serviceRegistry)  
            .getMetadataBuilder()  
            .build();  
  
        ③ return metadata.getSessionFactoryBuilder().build();  
    }  
    public static void shutdown() {  
        factory.close();  
    }  
    public static SessionFactory getSessionFactory() {  
        return factory;  
    }  
}
```

## Hibernate 5 Bootstrapping

- ① Create ServiceRegistry.
- ② Create Metadata.
- ③ Create SessionFactory.

### ServiceRegistry

- ServiceRegistry is interface.
- Some implementations are
  - ✓ StandardServiceRegistry,
  - ✓ BootstrapServiceRegistry,
  - ✓ EventListenerRegistry, ...
- Add, manage hibernate services.

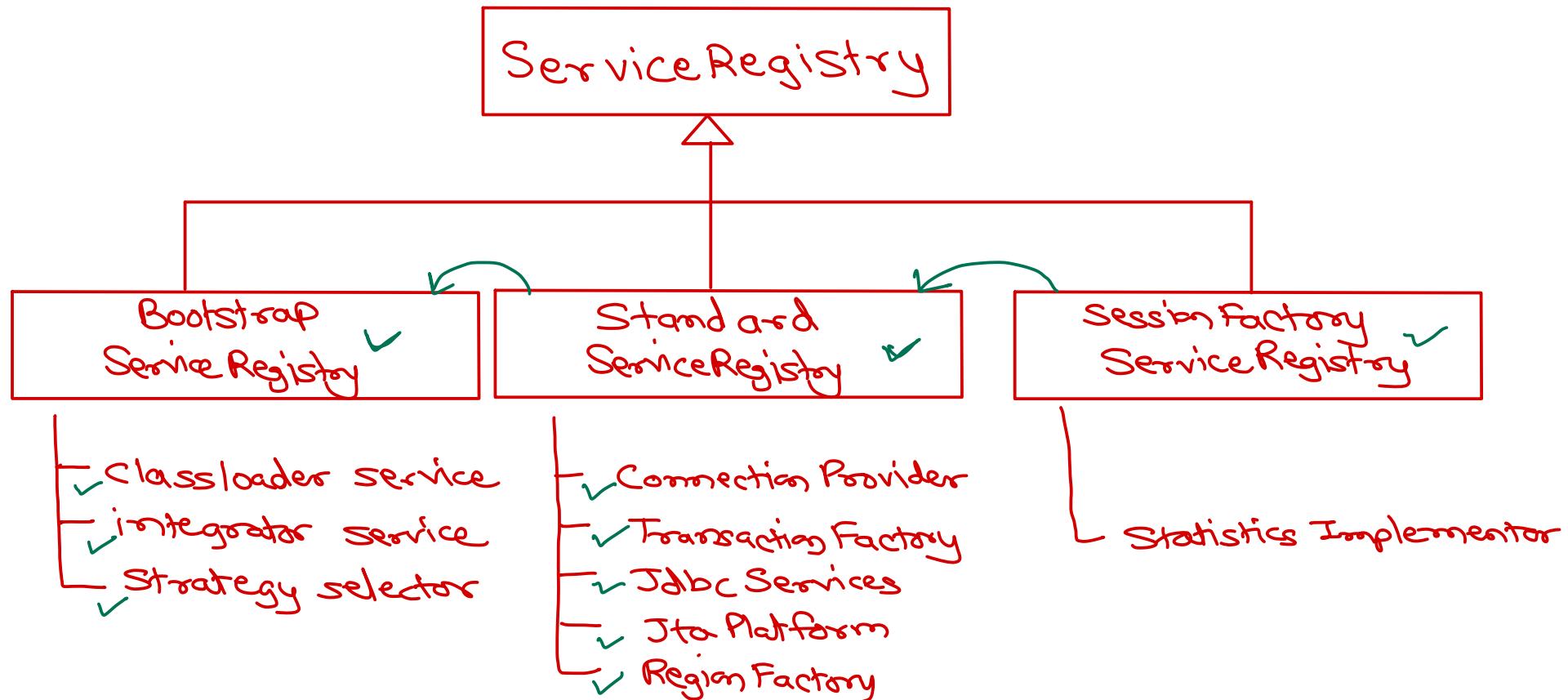
### Metadata

- Represents application's domain model & its database mapping. → ORM

<https://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html>



# Hibernate service registry



# Hibernate

- Hibernate3 added annotations for ORM.

- ORM using annotations

- @Entity
- @Table
- @Column
- @Id
- @Temporal
- @Transient

→ to convert date & time  
classes from java.sql  
to java.util.Date and/or  
java.util.Calendar.

Can be used to map  
Temporal.DATE → mysql  
DATE  
Temporal.TIME → mysql  
TIME  
Temporal.TIMESTAMP → mysql  
DATETIME  
TIMESTAMP

- @Column can be used on field level or on getter methods.



# Hibernate

hibernate.cfg.xml → <mapping> — />

- Earlier versions does ORM using .hbm.xml files.

```
@Entity  
@Table(name = "DEPT")  
public class Dept implements Serializable {  
    @Id //primary key  
    @Column(name = "deptno")  
    private int id; //deptno  
    @Column(name = "dname")  
    private String name; //dname  
    @Column(name = "loc")  
    private String loc; //loc  
    ...  
}
```

annotations

```
<hibernate-mapping>  
  <class name="com.sunbeam.sh.Dept" table="DEPT">  
    <id name="id" type="int">  
      <column name="DEPTNO" />  
      <generator class="assigned" />  
    </id>  
    <property name="name" type="java.lang.String">  
      <column name="DNAME" />  
    </property>  
    <property name="loc" type="java.lang.String">  
      <column name="LOC" />  
    </property>  
  </class>  
</hibernate-mapping>
```



# Hibernate Transaction

- In hibernate, autocommit is false by default.
- DML operations should be performed using transaction.
  - session.beginTransaction(): to start new tx.
  - tx.commit() & tx.rollback(): to commit/rollback tx.
- session.flush()
  - Forcibly synchronize in-memory state of hibernate session with database.
  - Each tx.commit() automatically flush the state of session.
  - Manually calling flush() will result in executing appropriate SQL queries into database.
  - Note that flush() will not commit the data into the RDBMS tables.
  - The flush mode can be set using session.setHibernateFlushMode(mode).
    - ALWAYS, AUTO, COMMIT, MANUAL
- If hibernate.connection.autocommit is set to true, we can use flush to force executing DML queries.



# openSession() vs getCurrentSession()

- openSession()
  - Create new hibernate session.
  - It is associated with JDBC connection (autocommit=false).
  - Can be used for DQL (get records), but cannot be used for DML without transaction.
  - Should be closed after its use.
- getCurrentSession()
  - Returns session associated with current context.
  - Session will be associated with one of the context (hibernate.current\_session\_context\_class).
    - thread: Session is stored in TLS.
    - jta: Session is stored in transaction-context given by JTA providers (like app servers).
    - custom: User implemented context.
  - This session is not attached with any JDBC connection.
  - JDBC connection is associated with it, when a transaction is created. The connection is given up, when transaction is completed.
  - The session is automatically closed, when scope is finished. It should not be closed manually.



# Using hibernate in Java EE application

- While dealing with single database, whole Java EE application should have single SessionFactory instance.
- This can be created as Singleton object in ServletContext. *(or as a singleton class HbUtil).*
- For each request new thread gets created.
- Hibernate session can be attached to thread session context, so that it can be accessed from any component.
- ORM is done using annotations on Entity objects and DAO layer is to be written using Hibernate (instead of JDBC).

Refer steps from notes.



# Spring Hibernate



# Spring Hibernate Integration *(Spring core)*

- Spring DI simplifies Hibernate ORM.
- LocalSessionFactoryBean provides session factory, while transaction automation is done by HibernateTransactionManager bean.
- Steps:
  - In pom.xml, add spring-orm, mysql-connector-java and hibernate-core.
  - Create dataSource, sessionFactory (with hibernate config), transactionManager beans. Also set default transactionManager. PackagesToScan ↗
  - Implement entity classes. Ensure that spring session factory config scan them.
  - Implement @Repository class and auto-wire session factory. Use factory.getCurrentSession() to obtain hibernate session and perform operations.
  - Implement @Service layer and mark business logic methods as @Transactional.
    - Note that single business operation (from service layer) may deal with multiple operations on different repositories. @Transactional put all ops under same tx.

*@Transactional can be used on @Repository. However using on @Service is recommended*



# Spring Boot Hibernate Integration

- Spring Boot Auto-configuration & DI simplifies Hibernate ORM.
- LocalSessionFactoryBean provides session factory, while transaction automation is done by HibernateTransactionManager bean.
- Steps:
  - Create spring boot project with Spring Data JPA and MySQL dependencies.
  - Define hibernate & database configurations into application.properties. *(or separate database.properties)*
  - Create a @Configuration class to create dataSource, sessionFactory and transactionManager beans. Mark it with @EnableTransactionManagement.
  - Implement entity classes with appropriate annotations.
  - Implement @Repository class and auto-wire session factory. Use factory.getCurrentSession() to obtain hibernate session and perform operations.
  - Implement @Service layer and mark business logic methods as @Transactional.
    - Note that single business operation (from service layer) may deal with multiple operations on different repositories. @Transactional put all ops under same tx.

*@Transactional can be used on @Repository. However using on @Service is recommended.*



# Hibernate Query Language



# Native Queries and HQL

- Ad-hoc SQL queries (on tables) can be executed in hibernate directly in hibernate.
  - `NativeQuery q = session.createSQLQuery(sql);`
- Hibernate recommends using HQL for ad-hoc queries.
- These queries are on hibernate entities (not on tables).
  - `Query q = session.createQuery(hql);`
- HQL supports SELECT, DELETE, UPDATE operation.
- INSERT is limited to `INSERT INTO ... SELECT ...;`



# Hibernate – HQL

- **SELECT**
  - from Book b
  - from Book b where b.subject = :p\_subject
  - from Book b order by b.price desc
  - select distinct b.subject from Book b
  - select b.subject, sum(b.price) from Book b group by b.subject
  - select new Book(b.id, b.name, b.price) from Book b
- **DELETE**
  - delete from Book b where b.subject = :p\_subject
- **UPDATE**
  - update Book b set b.price = b.price + 50 WHERE b.subject = :p\_subject
- **INSERT**
  - insert into Book(id, name, price) select id, name, price from old\_books





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

# Java EE

## Agenda

- Hibernate docs
- Hibernate 5 Bootstrapping
- ORM
- Session context
- HQL
- Hibernate in Java EE
- Spring Hibernate Integration

## Hibernate Docs

- <https://docs.jboss.org/hibernate/orm/5.5/javadocs/>
- [https://docs.jboss.org/hibernate/orm/5.5/userguide/html\\_single/Hibernate\\_User\\_Guide.html](https://docs.jboss.org/hibernate/orm/5.5/userguide/html_single/Hibernate_User_Guide.html)

## Hibernate 5 Bootstrapping

- Bootstrapping --> Getting Started --> SessionFactory creation.

## ORM

- @Entity
- @Table --> ClassName -> TableName
  - If ClassName is same as TableName, then this is optional.
- @Column --> FieldName -> ColumnName

```
@Column(name="columnName")
private DataType fieldName;
```

- If Column name is same as fieldName, then need not to mention name attribute of @Column.

```
@Column
private DataType fieldName;
```

- In an @Entity class, if Column name is same as fieldName, no need to write even @Column.

```
private DataType fieldName;
```

- Every field written in @Entity class is treated as column in database table.

- If you want to add some field, that is not in table then mark it as @Transient.

```
@Entity  
@Table(name="books")  
class Book {  
    // all field names are same as column names, so no need to write @Column  
    @Id // primary key (compulsory)  
    private int id;  
    private String name;  
    private String author;  
    private String subject;  
    private double price;  
    @Transient // column is not present in table  
    private double gst;  
    // getter/setter -- follow java naming conventions  
}
```

- @Temporal will convert RDBMS Date Time types into java.util.Date, java.util.Calendar, java.util.LocalDate, java.util.LocalTime, etc.

- MySQL DATE --> @Temporal(TemporalType.DATE)
- MySQL TIME --> @Temporal(TemporalType.TIME)
- MySQL DATETIME --> @Temporal(TemporalType.TIMESTAMP) // DATE + TIME
- MySQL TIMESTAMP --> @Temporal(TemporalType.TIMESTAMP) // DATE + TIME

```
@Entity  
@Table(name="customers")  
public class Customer {  
    @Id  
    private int id;  
    private String name;  
    private String email;  
    private String mobile;  
    private String address;  
    private String password;  
    @Temporal(TemporalType.DATE)  
    private Date birth;  
}
```

- @Column, @Id, @Temporal, @Transient are written on fields or getter methods.

- If written on fields, hibernate initialize fields directly using reflection (e.g. session.get()).
- If written on getters, hibernate initialize fields using corresponding setter methods. If any logic is present in setter will be executed. Naming conventions must be followed for getter/setter.
- Standard practice is to use these annotations on getters.

## Session context

```
Session session = factory.openSession()
```

- Creates a new Hibernate session associated with a JDBC connection.
- Progammer must close the session (to avoid resource leak).
- Get SELECT access to database and for DML operations need to start a transaction.

```
Session session = factory.getCurrentSession()
```

- Get hibernate session associated with current context, if available.
- If no hibernate session is available in current context, create new session and attach it to the context.
- Returns the same hibernate session if called multiple times, in the same context.
- Session context
  - thread --> Hibernate session is attached to the current thread (Thread Local Storage).
  - jta --> Java Transaction API (supported by application servers like JBoss or frameworks like Open-JTA).
  - custom --> User defined (depends on implementation)
- <property name="hibernate.current\_session\_context\_class">thread</property>
- When session context is closed, session is automatically closed. Progammer should not close session explicitly.
- Must start Transaction for each database operation (SELECT + DML).

## HQL (Hibernate Query Language)

- Hibernate (Classes/Fields) is abstraction over JDBC/RDBMS (Tables/Columns).
- Basic CRUD
  - get() --> find by id -- SELECT (single)
  - persist() --> insert new record -- INSERT (single)
  - delete() --> delete the record -- DELETE (single)
  - update() --> update the record -- UPDATE (single)
- HQL query language enable find by non-id, find multiple records with custom where clause & order by, grouping & joins, DML on multiple records.
- HQL --> Entity (class) and SQL --> Table (RDBMS).

```
String hql = "...";
Query<EntityClass> q = session.createQuery(hql);
// execute the query -- q.getResultList(), q.getSingleResult(),
q.executeUpdate().
```

## HQL queries

### SELECT operation

- default operation
- "from EntityClass alias"

- Default operation is SELECT.
- Default fetch all columns mapped in EntityClass (@Column).
- from EntityClass alias --> Hibernate --> SELECT col1, col2, ... FROM RdbmsTable alias.
- "from Customer c" --> get all customers
- "from Book b" --> get all books
- "from EntityClass alias where condition"
  - Default operation is SELECT.
  - Default fetch all columns mapped in EntityClass (@Column).
  - from EntityClass alias --> Hibernate --> SELECT col1, col2, ... FROM RdbmsTable alias WHERE condition.
  - "from Customer c where c.email = :p\_email"
    - ?1 (numbered parameter) OR :p\_email (named parameter)
      - q.setParameter("p\_email", email);
      - q.getSingleResult()
    - JDBC: SELECT \* FROM customers c WHERE c.email = ?
      - stmt.setString(1, email)
      - rs = stmt.executeQuery()
  - "from Book b where b.subject = :p\_subject"
    - p\_subject (named parameter)
      - q.setParameter("p\_subject", subject)
      - q.getResultList()
- SELECT Examples:
  - "from Customers c order by c.address"
  - "from Book b where b.subject = :p\_subject order by b.name desc"
- Advanced SELECT Examples:
  - "select new Book(id, name, price) FROM Book b"
    - SQL --> SELECT id, name, price FROM books b
    - Selected columns will be picked into Book object. Book class MUST have a constructor with three argument (id, name and price).
  - "select b.subject, SUM(b.price) FROM Book b group by b.subject"
    - SQL --> SELECT b.subject, SUM(b.price) FROM books b GROUP BY b.subject
    - List<Object[]> list = q.getResultList();
      - To collect the result of arbitrary queries (any columns -- not mapped in Entity) use Object[]
      - Object[] will have number of columns fetched in the query.
      - In this example: Object[] will be array of two elements (Object)
        - 0th element = subject
        - 1st element = sum/total price

## UPDATE operation

- To update single record for given ID, use session.update()
- To update multiple rows use HQL.
- "update Book b set b.price = b.price + b.price \* 0.05 where b.subject = :p\_subject"

```
String hql = "update Book b set b.price = b.price + b.price * 0.05 where  
b.subject = :p_subject";  
Query q = session.createQuery(hql);  
q.setParameter("p_subject", subject);  
int count = q.executeUpdate();
```

## DELETE operation

- To delete single record for given ID, use session.delete() or session.remove()
- To delete multiple rows use HQL.
- "delete from Book b where b.subject = :p\_subject"

```
String hql = "delete from Book b where b.subject = :p_subject";  
Query q = session.createQuery(hql);  
q.setParameter("p_subject", subject);  
int count = q.executeUpdate();
```

## INSERT operation

- To insert single record use session.save() or session.persist()
- HQL insert only support INSERT INTO ... SELECT FROM kind of operations.

## SQL

- Hibernate support writing SQL queries, but not recommended.

```
String sql = "SELECT DISTINCT subject FROM books";  
NativeQuery<String> q = session.createNativeQuery(sql);  
return q.getResultList();
```

## Hibernate in Java EE (Optional Assignment)

- Do following steps in MVC Maven Bookshop assignment
- Delete old (JDBC) daos, pojos and utils i.e. BookDao, CustomerDao, Book, Customer, DbUtil, jdbc.properties.
- Into pom.xml add hibernate-core maven dependency.
- Copy hibernate.cfg.xml into resources from demo10.
- Copy HbUtil, Book, Customer, BookDaoImpl, CustomerDaoImpl in appropriate packages from demo10.
- Modify Java beans to accomodate new DAO class (change class name & method names as appropriate).
- Run application.
- Refer slides for the concepts.

## Spring Boot Hibernate Integration

- Refer slides for the concepts.

### Demo 11 (Console application)

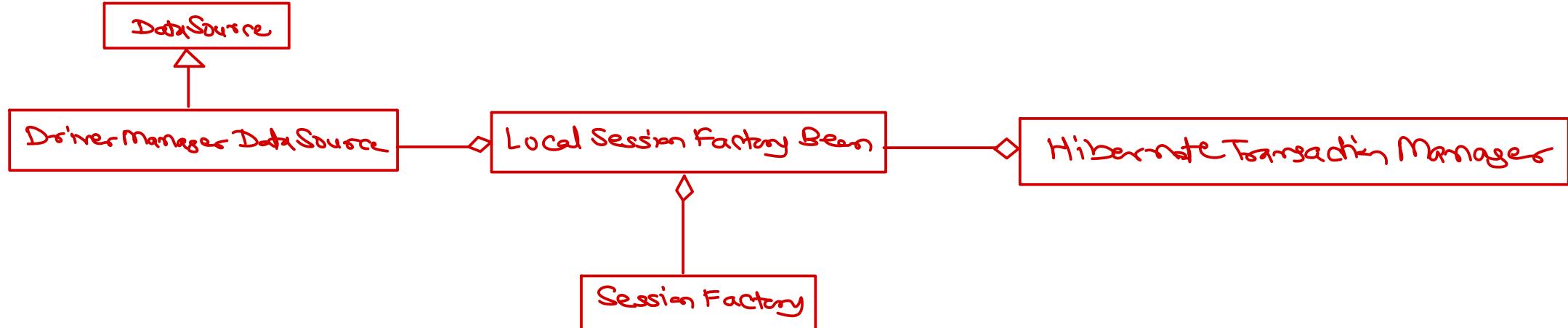
- step 1. New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Data JPA
    - MySQL Driver
  - Finish
- step 2. Create database.properties file. Add database & hibernate properties in it.
- step 3. Create HibernateConfig class.
  - Mark it @Configuration, @EnableTransactionManagement and @PropertySource("classpath:database.properties")
  - Add fields to get values from properties file using @Value.
  - Create beans (@Bean) dataSource, sessionFactory and transactionManager.
- step 4. Create Customer and Book pojo class. Add appropriate ORM annotations.
- step 5. Create BookDaoImpl and CustomerDaoImpl classes. Mark them as @Repository.
- step 6. @Autowired dao classes in main class and test the dao methods.



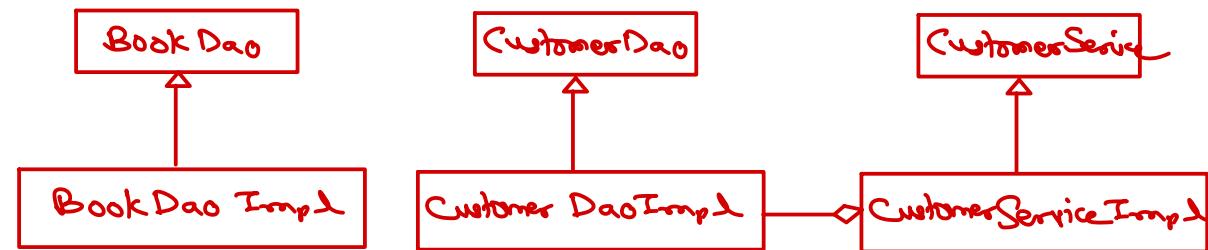
# Java EE

Trainer: Nilesh Ghule





# Spring Hibernate



# Spring Boot Hibernate Integration

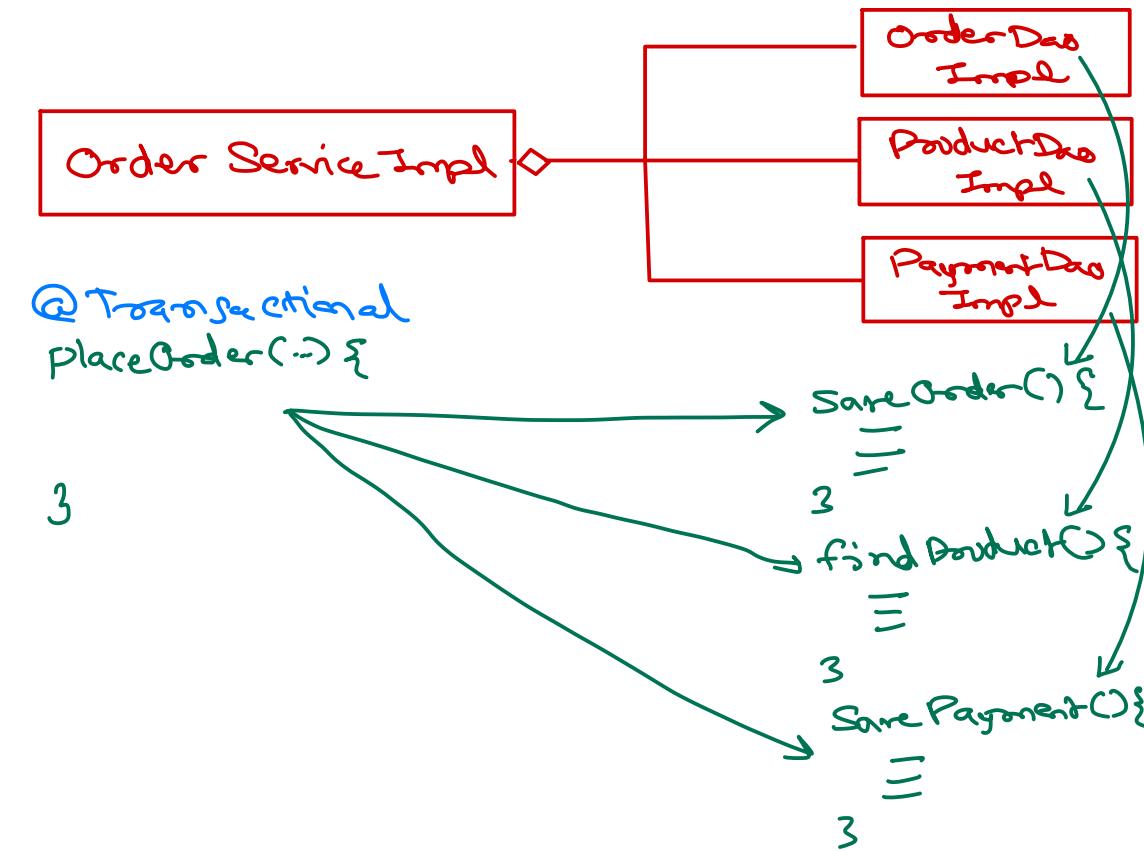
- Spring Boot Auto-configuration & DI simplifies Hibernate ORM.
- LocalSessionFactoryBean provides session factory, while transaction automation is done by HibernateTransactionManager bean.
- Steps:
  - Create spring boot project with Spring Data JPA and MySQL dependencies.
  - Define hibernate & database configurations into application.properties. *(or separate database.properties)*
  - Create a @Configuration class to create dataSource, sessionFactory and transactionManager beans. Mark it with @EnableTransactionManagement.
  - Implement entity classes with appropriate annotations.
  - Implement @Repository class and auto-wire session factory. Use factory.getCurrentSession() to obtain hibernate session and perform operations.
  - Implement @Service layer and mark business logic methods as @Transactional.
    - Note that single business operation (from service layer) may deal with multiple operations on different repositories. @Transactional put all ops under same tx.

*@Transactional can be used on @Repository. However using on @Service is recommended.*



# @Service layer

- @Repository layer contains database operations (i.e. CRUD operations, ...).
- @Service layer contains business logic. It is implemented as per business operations.
- One @Service component may have one or more DAO component dependencies.
- It is common practice to handle transactions in service layer.



# @Transactional



- @Transactional is declarative transaction management of Spring.
- It can be used method level or class level.  
If used on class level, it applies to all methods in the class.
- Spring internally use JDBC transaction in AOP fashion.
  - ✓ start transaction (before method).
  - ✓ commit transaction (if method is successful).
  - ✓ rollback transaction (if method throw exception).
- Transaction management is done by platform transaction manager e.g. datasource, hibernate or jpa transaction manager.



# @Transactional

- If one transactional method invokes another transactional method, transaction behaviour is defined by propagation attribute.

- REQUIRED: Support a current transaction, create a new one if none exists. (default)
- REQUIRES\_NEW: Create a new transaction, and suspend the current transaction if one exists.
- SUPPORTS: Support a current transaction, execute non-transactionally if none exists.
- MANDATORY: Support a current transaction, throw an exception if none exists.
- NEVER: Execute non-transactionally, throw an exception if a transaction exists.
- NOT\_SUPPORTED: Execute non-transactionally, suspend the current transaction if one exists.
- NESTED: Execute within a nested transaction (save points) if a current transaction exists, behave like REQUIRED otherwise.

## Customer Service Impl

```
@Transactional  
authenticate(-) {  
    c = findByEmail();  
    ...  
    ...  
    3
```

  ↓

```
@Transactional  
findByEmail(-) {  
    ...  
    ...  
    3
```

```
@Transactional (isolation = ...)  
    ↓  
    jdbc → Con.setIsolationLevel(...);  
    ↓  
    RDBMS → ACID + x
```

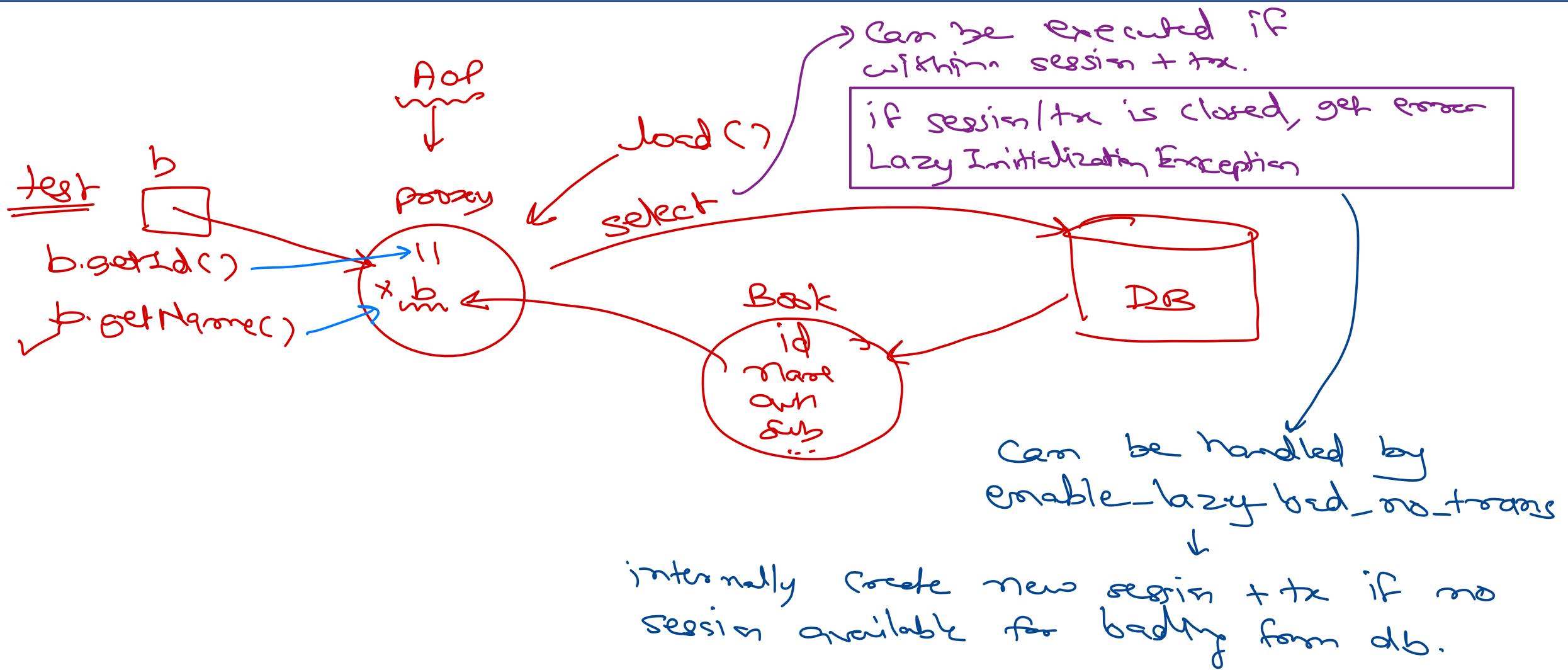


# Hibernate CRUD



- Hibernate Session methods

- get() or find(): Find the database record by primary key and return it. If record is not found, returns null.
- load(): Returns proxy for entity object (storing only primary key). When fields are accessed on proxy, SELECT query is fired on database and record data is fetched. If record not found, exception is thrown.
- save(): Assign primary key to the entity and execute INSERT statement to insert it into database. Return primary key of new record. *return Serializable (PK)*
- persist(): Add entity object into hibernate session. Execute INSERT statement to insert it into database (for all insertable columns) while committing the transaction. *session void*
- update(): Add entity object into hibernate session. Execute UPDATE statement to update it into database while committing the transaction. All (updateable) fields are updated into database (for primary key of entity).
- saveOrUpdate() or merge(): Execute SELECT query to check if record is present in database. If found, execute UPDATE query to update record; otherwise execute INSERT query to insert new record.
- delete() or remove(): Delete entity from database (for primary key of entity) while committing the transaction.
- evict() or detach(): Removes entity from hibernate session. Any changes done into the session after this, will not be automatically updated into the database.
- clear(): Remove all entity objects from hibernate session.
- refresh(): Execute SELECT query to re-fetch latest record data from the database.

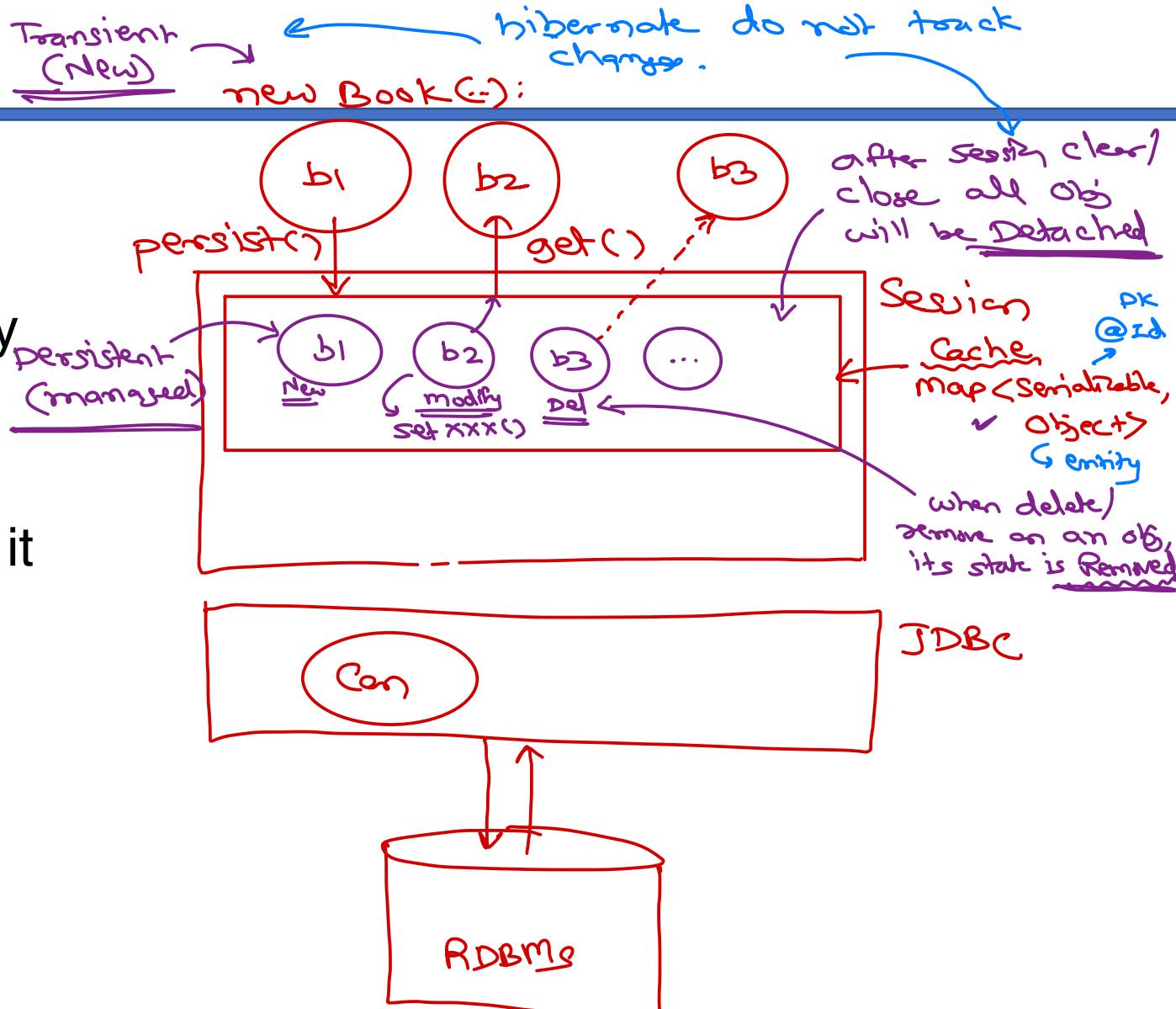


# Hibernate Entity life cycle



# Hibernate session cache

- Collection of entities per session – persistent objects.
- Hibernate keep track of state of entity objects and update into database.
- Session cache cannot be disabled.
- If object is present in session cache, it is not searched into session factory cache or database.
- Use refresh() to re-select data from the database forcibly.



# Hibernate – Entity life cycle

- Transient

- New Java object of entity class.
- This object is not yet associated with hibernate.

- Persistent

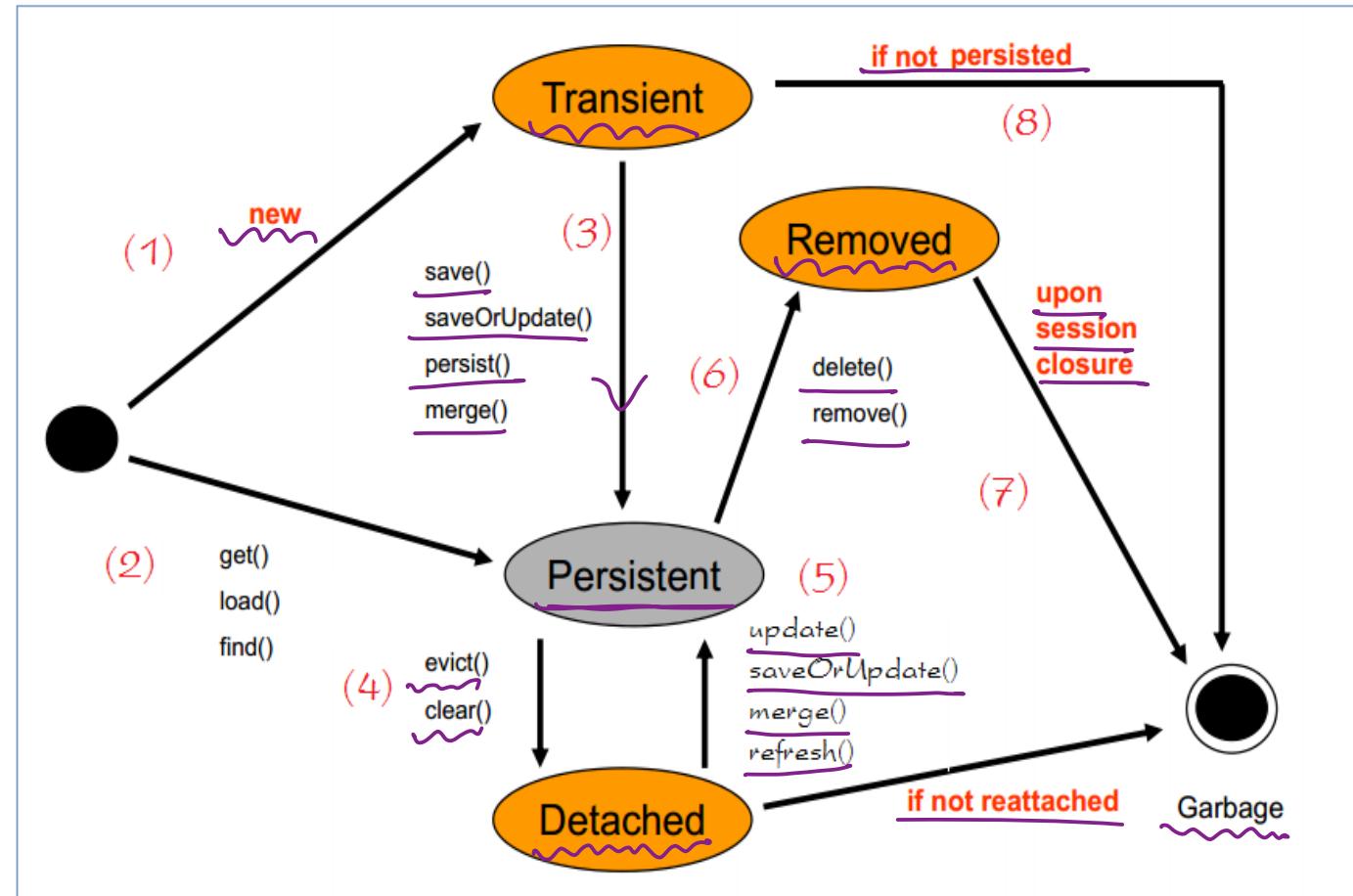
- Object in session cache.
- For all objects created by hibernate or associated with hibernate.
- State is tracked by hibernate and updated in database during commit.

- Never garbage collected.

- Detached
- Object removed from session cache.

- Removed

- Object whose corresponding row is deleted from database.



# Hibernate Entity associations



# Hibernate Relations

- RDBMS Relations

- RDBMS tables are designed by process of Normalization.
- To avoid redundancy, data is organized into multiple tables.
- These tables are related to each other by means of primary key and foreign key constraints.
- ER diagram shows relations among the tables.
- The relations can be:
  - OneToOne ✓
  - OneToMany ✓
  - ManyToOne ✓
  - ManyToMany ✓

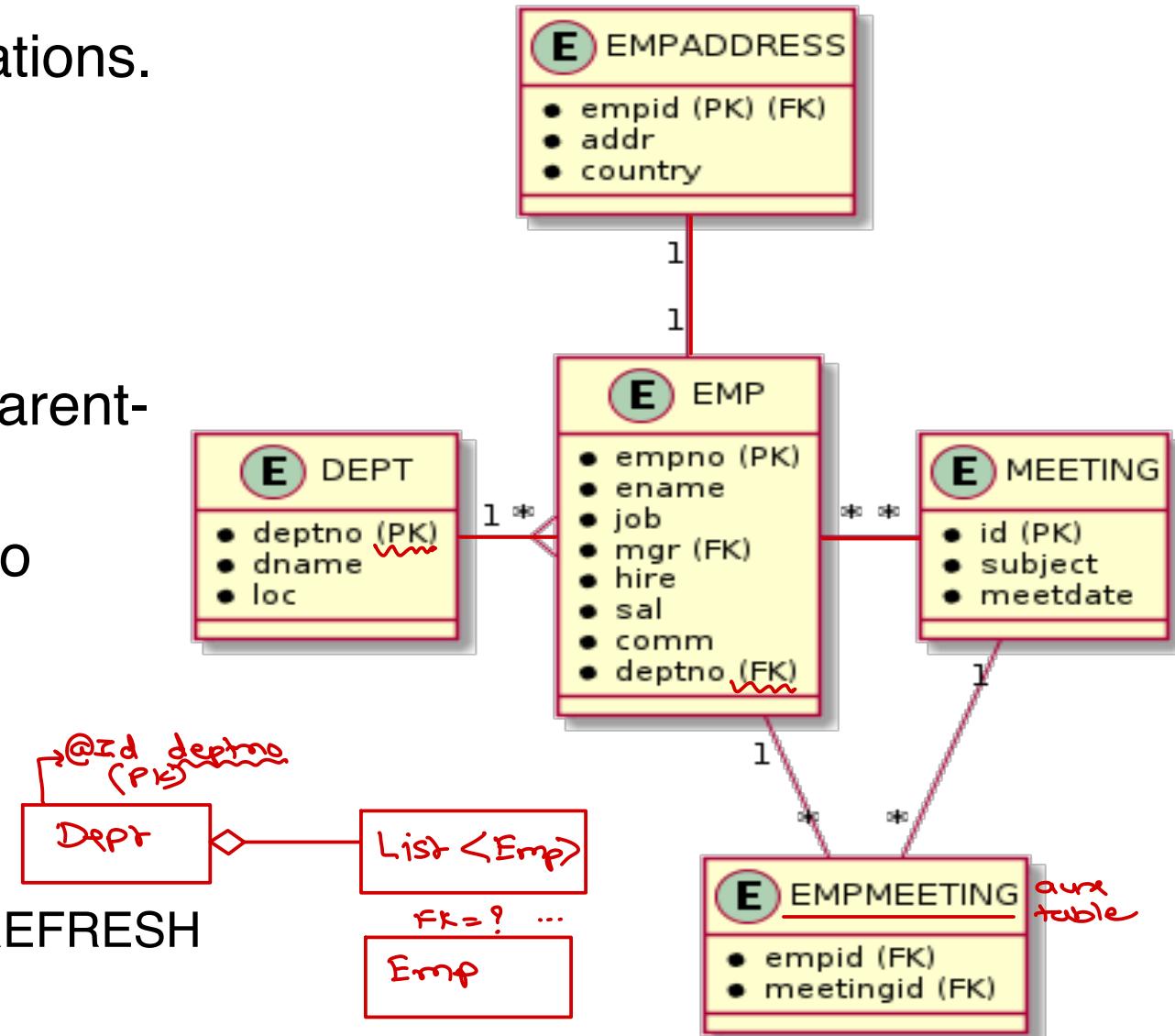
- Hibernate Relations

- Hibernate is ORM tool.
- Java classes/objects are related by means of inheritance and associations.
- Hibernate maps RDBMS table relations to Java class/object relations.
- Hibernate supports both relations
  - Associations ✓
  - Inheritance
- Associations are supported with set of annotations e.g. @OneToMany, @ManyToOne, @ManyToMany, @OneToOne
- Inheritance is supported with set of annotations e.g. @MappedSuperclass, @Inheritance.



# Hibernate Relations (associations)

- Hibernate represents RDBMS table relations.
  - OneToOne
  - OneToMany
  - ManyToOne
  - ManyToMany
- OneToMany & ManyToOne represent parent-child relation between tables.
- Primary key of parent table is mapped to foreign key of child table.
- FetchType
  - Lazy or Eager
- CascadeType
  - PERSIST, MERGE, DETACH, REMOVE, REFRESH



# @OneToMany (uni-directional)

- A Dept has Many Emp.
  - mappedBy – foreign key field in Emp table (that reference to primary key of Dept table).
  - FetchType
    - LAZY: Fetch Dept only (simple SELECT query) and fetch Emp only when empList is accessed (simple SELECT query with WHERE clause on deptno)
    - EAGER: Fetch Dept & Emp data in single query (OUTER JOIN query)
  - CascadeType
    - PERSIST: insert Emp in list while inserting Dept (persist())
    - REMOVE: delete Emp in dept while deleting Dept (remove())
    - DETACH: remove Emp in dept from session while removing Dept from session (detach())
    - REFRESH: re-select Emp in dept while re-selecting Dept (refresh())
    - MERGE: add Emp in dept intion session while adding Dept into session (merge())

```
@Entity @Table(name="dept")
class Dept {
    @Id
    @Column private int deptno; ✓
    @Column private String dname; ✓
    @Column private String loc; ✓
    @OneToMany(mappedBy="deptno")
    private List<Emp> empList;
    // ...
}

@Entity @Table(name="emp")
class Emp {
    @Id
    @Column private int empno; ✓
    @Column private String ename; ✓
    @Column private double sal; ✓
    @Column private int deptno; ✓
    // ...
}
```

FK field





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>



# Java EE

Trainer: Nilesh Ghule



# @OneToMany (uni-directional)

- A Dept has Many Emp.
  - mappedBy – foreign key field in Emp table (that reference to primary key of Dept table).
  - FetchType
    - LAZY: Fetch Dept only (simple SELECT query) and fetch Emp only when empList is accessed (simple SELECT query with WHERE clause on deptno)
    - EAGER: Fetch Dept & Emp data in single query (OUTER JOIN query)
  - CascadeType
    - PERSIST: insert Emp in list while inserting Dept (persist())
    - REMOVE: delete Emp in dept while deleting Dept (remove())
    - DETACH: remove Emp in dept from session while removing Dept from session (detach())
    - REFRESH: re-select Emp in dept while re-selecting Dept (refresh())
    - MERGE: add Emp in dept intion session while adding Dept into session (merge())

```
@Entity @Table(name="dept")
class Dept {
    @Id
    @Column private int deptno; ✓
    @Column private String dname; ✓
    @Column private String loc; ✓
    @OneToMany(mappedBy="deptno")
    private List<Emp> empList;
    // ...
}
.

@Entity @Table(name="emp")
class Emp {
    @Id
    @Column private int empno; ✓
    @Column private String ename; ✓
    @Column private double sal; ✓
    @Column private int deptno; ✓
    // ...
}
```

The diagram illustrates the unidirectional relationship between the Dept and Emp entities. It shows the annotated code for both classes. A blue circle highlights the `@OneToMany` annotation in the `Dept` class. A red box highlights the `deptno` column in the `Emp` class. A blue arrow points from the `deptno` column in the `Emp` class to the `@OneToMany` annotation in the `Dept` class, indicating it is the mappedBy field. A red line with the label "FK field" connects the `deptno` column in the `Emp` class to the `deptno` column in the `Dept` class, representing the foreign key relationship.



## @ManyToOne (uni-directional)

- Many Emp can have same Dept.
  - FetchType – LAZY or EAGER *\* ← default*
  - CascadeType - PERSIST, MERGE, DETACH, REMOVE, REFRESH
  - @JoinColumn is used along with @ManyToOne to specify foreign key column in EMP table (that reference to primary key of DEPT table).

```
Emp e = session.get(Emp.class, 7900);
System.out.println(e);
System.out.println(e.getDept());
```

```
@Entity @Table(name="dept")
class Dept {
    @Id PK
    @Column private int deptno;
    @Column private String dname;
    @Column private String loc;
    // ...
}
```

```
@Entity @Table(name="emp")
class Emp {
    @Id
    @Column private int empno;
    @Column private String ename;
    @Column private double sal;
    @Column private int deptno;
    @ManyToOne eager load by default.
    @JoinColumn(name="deptno")
    private Dept dept;
    // ...
}
```

*Fk Column  
in Emp table*



# @OneToMany and @ManyToOne (bi-directional)

- A Dept have many Emps.
- Many Emp can have same Dept.
- @ManyToOne in Emp class
  - Use @JoinColumn to specify FK column in EMP table.
- @OneToMany in Dept class
  - Use mappedBy to specify FK field in Emp class – now declared as Dept object.
  - FK value is taken from inner Dept object's @Id field.

```
@Entity @Table(name="DEPT")
class Dept {
    @Id
    @Column private int deptno;
    @Column private String dname;
    @Column private String loc;
    @OneToMany(mappedBy="dept")
    private List<Emp> empList;
}
```

```
@Entity @Table(name="EMP")
class Emp {
    @Id
    @Column private int empno;
    @Column private String ename;
    @Column private double sal;
    @Column private int deptno; X
}
```

```
@ManyToOne
@JoinColumn(name="deptno")
private Dept dept;
```





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

# Java EE

## Agenda

- Spring Hibernate Quick Revision
- Hibernate entity life cycle
- Hibernate entity relations
- Auto-generated Primary key

## Spring Hibernate Quick Revision

- spring-orm is wrapper on hibernate and provides util beans like DataSource, LocalSessionFactoryBean and HibernateTransactionManager.
  - DataSource: javax.sql.DataSource
    - Any class implemented from this interface is a valid DataSource bean e.g. DriverMangerDataSource (wrapper on JDBC), BasicDataSource (tomcat connection pool), c3p0 hibernate data source, Hikari CP, etc.
  - SessionFactory: org.hibernate.SessionFactory
    - Hibernate has internal implementation of SessionFactory (in bare Hibernate created in HbUtil).
    - Spring provides LocalSessionFactoryBean class as wrapper on SessionFactory. It is used as singleton bean to create SessionFactory.
      - hibernateProperties --> all hibernate related settings (hibernate.cfg.xml -- hibernate.xxx).
      - dataSource --> database connection
      - packagesToScan --> list of packages to be scanned for hibernate entities @Entity.
  - TransactionManger:
    - In hibernate Transaction is mandetory.
    - Spring provides HibernateTransactionManager that automates transation management.
      - sessionFactory --> so that TransactionManager can get the session reference internally (factory.getCurrentSession())

### @Transactional

- Without spring

```
try {  
    //beginTransaction()  
    dao.method();  
    //commit()  
} catch(Exception ex) {  
    //rollback  
}
```

- Spring provides @Transactional annotation. If this annotation is used on any method, spring does automate transaction management.
  - Internally it calls beginTransaction before method is called. // pre-processing
  - After the method it calls commit, if method is successful. // post-processing
  - After the method it calls rollback, if method throws any exception. // post-processing
  - Spring use AOP for implementing this.
- If @Transactional is used on a class, it applies to each method in the class.
- Refer slides.

## Hibernate Session Cache

- Refer slides

## Hibernate CRUD (Session interface)

- get():
  - Hibernate method to find by ID (PK) -- WHERE id = ?
  - Eager loading (Immediately execute SELECT query, populate object and return it)
- find():
  - JPA compliant method
  - Same as get()
- load():
  - Hibernate method to find by ID (PK) -- WHERE id = ?
  - Lazy loading
    - Returns entity proxy (containing id)
    - When other fields are accessed, execute SELECT query and populate object.
    - If transaction is closed at that time, then throw LazyInitializationException.
    - This can be resolved by creating new session & transaction, when fields are accessed. For this set hibernate.enable\_lazy\_load\_no\_trans=true.
- save()
  - Hibernate method to insert a new row in database.
  - Auto-generate ID and execute INSERT query on database.
  - Returns Primary key of new row.
- persist()
  - JPA compliant method.
  - Execute INSERT query on database while committing transaction.
- saveOrUpdate()
  - Hibernate method to insert or update row in database.
  - First execute SELECT query to check if row is present in database (for given id).
  - If present, execute UPDATE query; otherwise execute INSERT query()
- merge()
  - JPA compliant method
  - Similar to saveOrUpdate()
- delete()
  - Hibernate method to delete the row from the database.
- remove()
  - JPA compliant method

- Same as delete()
- evict()
  - Hibernate method to remove the object from the session cache.
- detach()
  - JPA compliant method
  - Same as evict()
- clear()
  - Removes all objects from session cache.

## Hibernate entity life cycle

- Refer slides for details.
- Transient
  - Also called as NEW in JPA
- Persistent
  - Also called as MANAGED in JPA
- Detached
- Removed

## Demo 12 (Service layer and @Transactional)

- step 1. New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Data JPA
    - MySQL Driver
  - Finish
- step 2. Create database.properties file. Add database & hibernate properties in it.
- step 3. Create HibernateConfig class.
  - Mark it @Configuration, @EnableTransactionManagement and  
@PropertySource("classpath:database.properties")
  - Add fields to get values from properties file using @Value.
  - Create beans (@Bean) dataSource, sessionFactory and transactionManager.
- step 4. Create Customer and Book pojo class. Add appropriate ORM annotations.
- step 5. Create BookDao and CustomerDao interface.
- step 6. Create BookDaoImpl and CustomerDaoImpl classes. Mark them as @Repository. Do not use @Transactional here.
- step 7. Create BookService (Lab assignment) and CustomerService interface.
- step 8. Create BookServiceImpl (Lab assignment) and CustomerDaoImpl classes. Mark them as @Service and @Transactional. @Autowired dao classes and invoke the dao methods.
- step 9. In main class @Autowired CustomerService and BookService (LabAssignment) and test the methods.

## Demo 12 (Hibernate Quick Testing)

- Demo12 continued
- For quick testing of basic hibernate concepts, create TestRepository class. Mark it with @Repository and @Transactional.

- Add test methods in the class e.g. testFindBook(), testLoadBook(), ...
- NOTE: hibernate.enable\_lazy\_load\_no\_trans = true into HibernateConfig while testing load() method to remove LazyInitializationException.
- @Autowired TestRepository into main class and call these test methods.

## Demo 13 (OneToMany relation)

- New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Data JPA
    - MySQL Driver
  - Finish
- Create database.properties file. Add database & hibernate properties in it.
  - Also add hibernate.enable\_lazy\_load\_no\_trans = true.
- Create HibernateConfig class.
  - Mark it @Configuration, @EnableTransactionManagement and @PropertySource("classpath:database.properties")
  - Add fields to get values from properties file using @Value.
  - Create beans (@Bean) dataSource, sessionFactory and transactionManager.
- Create Dept and Emp pojo class. Add appropriate ORM annotations.
- In Dept class add "empList" field with annotation @OneToMany(mappedBy="deptno") and corresponding getter/setter methods.
- Write a test Repository class EmpDeptDaoTest. Mark class with @Repository and @Transactional. @Autowired it into main class.
- Implement test method testGetDept() in EmpDeptDaoTest. Invoke method from main class. Observe SQL queries generated. Two queries are executed (for Dept and Emp). Note that default fetch type is "lazy" for OneToMany.
- In Dept class change @OneToMany annotation to add fetch=FetchType.EAGER. Run application again. Observe SQL queries generated. Only a single JOIN query is executed.
- In Dept class, in constructors initialize empList field to empty ArrayList. Also add a method addEmp().
- Implement test method testAddDept() in EmpDeptDaoTest. Invoke method from main class. Observe SQL queries generated. By default only Dept is inserted. (Delete it manually from MySQL prompt).
- In Dept class change @OneToMany annotation to add cascade=CascadeType.PERSIST. Run application again. Observe SQL queries generated. Dept and Emps are inserted now.
- Implement test method testDelDept() in EmpDeptDaoTest. In Dept class change @OneToMany annotation to add cascade=CascadeType.REMOVE. Invoke testDelDept() method from main class. Observe SQL queries generated. Emps and Dept are deleted.

## CascadeType types

- PERSIST
  - When parent entity is inserted/persisted (session.persist()), then all child entities in it are also inserted/persisted (session.persist())
- REMOVE
  - When parent entity is deleted/removed (session.remove()), then all child entities in it are also deleted/removed (session.remove()). To avoid orphan rows/entities, child entities are deleted

before parent entities.

- MERGE
  - When parent entity is added into the session cache (session.merge()), then all child entities in it are also added into the session cache (session.merge())
- DETACH
  - When parent entity is removed from the session cache (session.detach()), then all child entities in it are also removed from the session cache (session.detach())
- REFRESH
  - When parent entity is re-selected from database (session.refresh()), then all child entities in it are also re-selected from database (session.refresh())
- ALL
  - combination of all above.

## Demo 14 (ManyToOne and OneToMany relation - bi-directional)

- New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Data JPA
    - MySQL Driver
  - Finish
- Create database.properties file. Add database & hibernate properties in it.
  - Also add hibernate.enable\_lazy\_load\_no\_trans = true.
- Create HibernateConfig class.
  - Mark it @Configuration, @EnableTransactionManagement and @PropertySource("classpath:database.properties")
  - Add fields to get values from properties file using @Value.
  - Create beans (@Bean) dataSource, sessionFactory and transactionManager.
- Create Dept and Emp pojo class. Add appropriate ORM annotations.
- In Emp class add "dept" field with annotation @ManyToOne and @JoinColumn(name="deptno"). Here "deptno" is Foreign key "column" name in the "emp" table.
- In Dept class add "empList" field with annotation @OneToMany(mappedBy="dept") and corresponding getter/setter methods. Here "dept" is field name in the Emp class.
- Write a test Repository class DeptEmpDaoTest. Mark class with @Repository and @Transactional. @Autowired it into main class.
- Implement test method testGetEmp() in DeptEmpDaoTest. Invoke method from main class. Observe SQL queries generated. Only single JOIN query is executed. Note that default fetch type is "eager" for ManyToOne.
- In Emp class change @ManyToOne annotation to add fetch=FetchType.LAZY. Run application again. Observe SQL queries generated. Two queries are executed (for Emp and its Dept).
- Implement test method testGetDept() in DeptEmpDaoTest. Set @OneToMany fetch=LAZY in Dept class and @ManyToOne fetch=EAGER in Emp class. Invoke method from main class. Observe SQL queries generated. Two queries are executed (for Dept and Emp).

# Hibernate - Primary keys



# Auto-generate Primary keys

- @GeneratedValue annotation is used to auto-generate primary key.
- This annotation is used with @Id column.
- There are different strategies for generating ids.
  - AUTO: Depends on database dialect.
    - MySQL 8: id will be taken from "next\_val" column of "gen" table.
  - IDENTITY: RDBMS AUTO\_INCREMENT / IDENTITY ← MS-SQL Server
    - MySQL 8: id will be AUTO\_INCREMENT
  - TABLE: Dedicated table for PK generation of all entities
    - MySQL 8: @TableGenerator(name="gen", initialValue=1000, pkColumnName="book\_ids", valueColumnName="id", table="id\_gen", allocationSize=1)
  - SEQUENCE: RDBMS sequence using @SequenceGenerator
    - MySQL 8: Emulated with table.



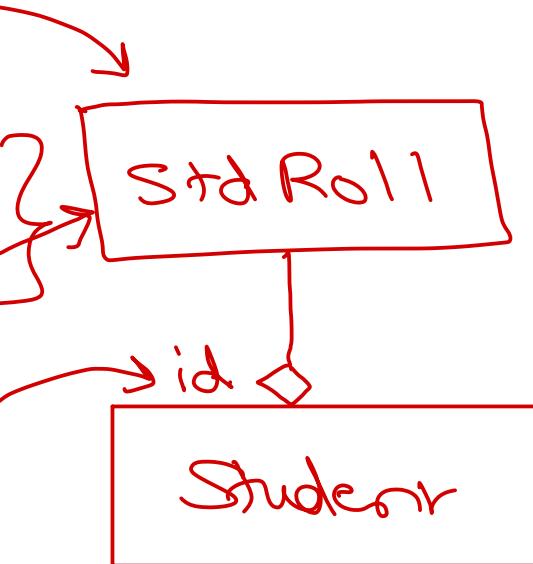
# Auto-generate Primary keys

- `@GenericGenerator` is used to specify config of hibernate's id generation strategies.
- Hibernate allows few more strategies.
  - native: depends on db/dialect.
  - guid: 128-bit unique id
  - hilo: id gen using high low algo.
  - identity: auto incr or identity col.
  - sequence: db seq.
  - increment: select  $\text{max}(\text{id})+1$  from table.
  - foreign: id corresponding to foreign key `@PrimaryKeyJoinColumn`.



# Composite Primary key

- Hibernate @Id is always **serializable**.
- In case of composite PK, create a new class for PK and mark as @Embeddable.
- Create object of that class into entity class & mark it as @EmbeddedId.
- @Embeddable class must implement equals() and hashCode().

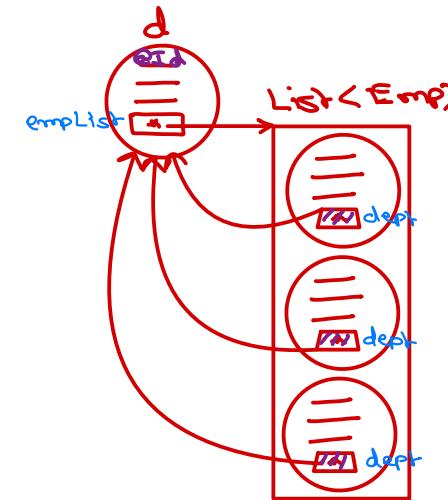


# Hibernate Entity associations



# @OneToMany and @ManyToOne (bi-directional)

- A Dept have many Emps.
- Many Emp can have same Dept.
- @ManyToOne in Emp class
  - Use @JoinColumn to specify FK column in EMP table.
- @OneToMany in Dept class
  - Use mappedBy to specify FK field in Emp class – now declared as Dept object.
  - FK value is taken from inner Dept object's @Id field.



```
@Entity @Table(name="DEPT")
class Dept {
    @Id
    @Column private int deptno;
    @Column private String dname;
    @Column private String loc;
    @OneToMany(mappedBy="dept")
    private List<Emp> empList;
}
```

```
@Entity @Table(name="EMP")
class Emp {
    @Id
    @Column private int empno;
    @Column private String ename;
    @Column private double sal;
    @Column private int deptno; X X X
```

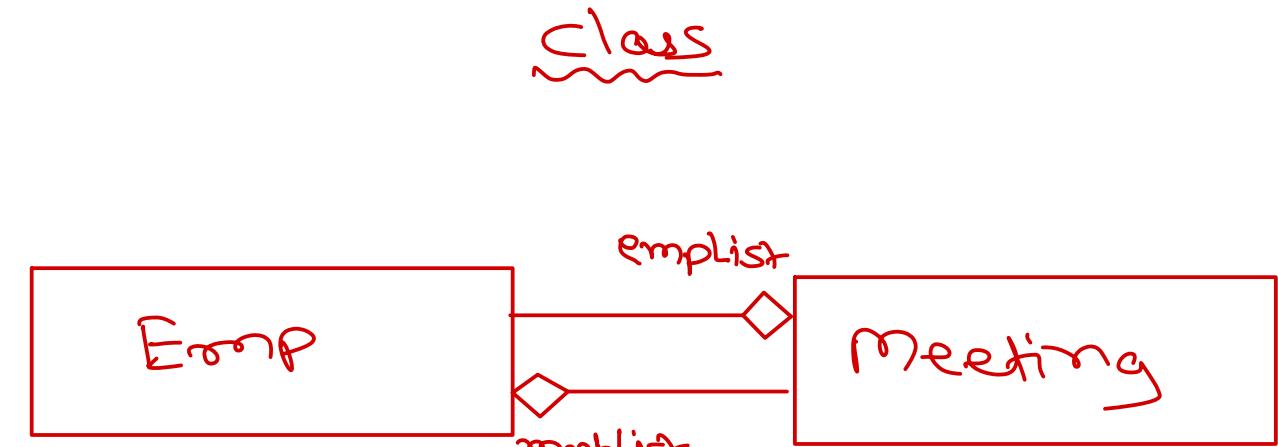
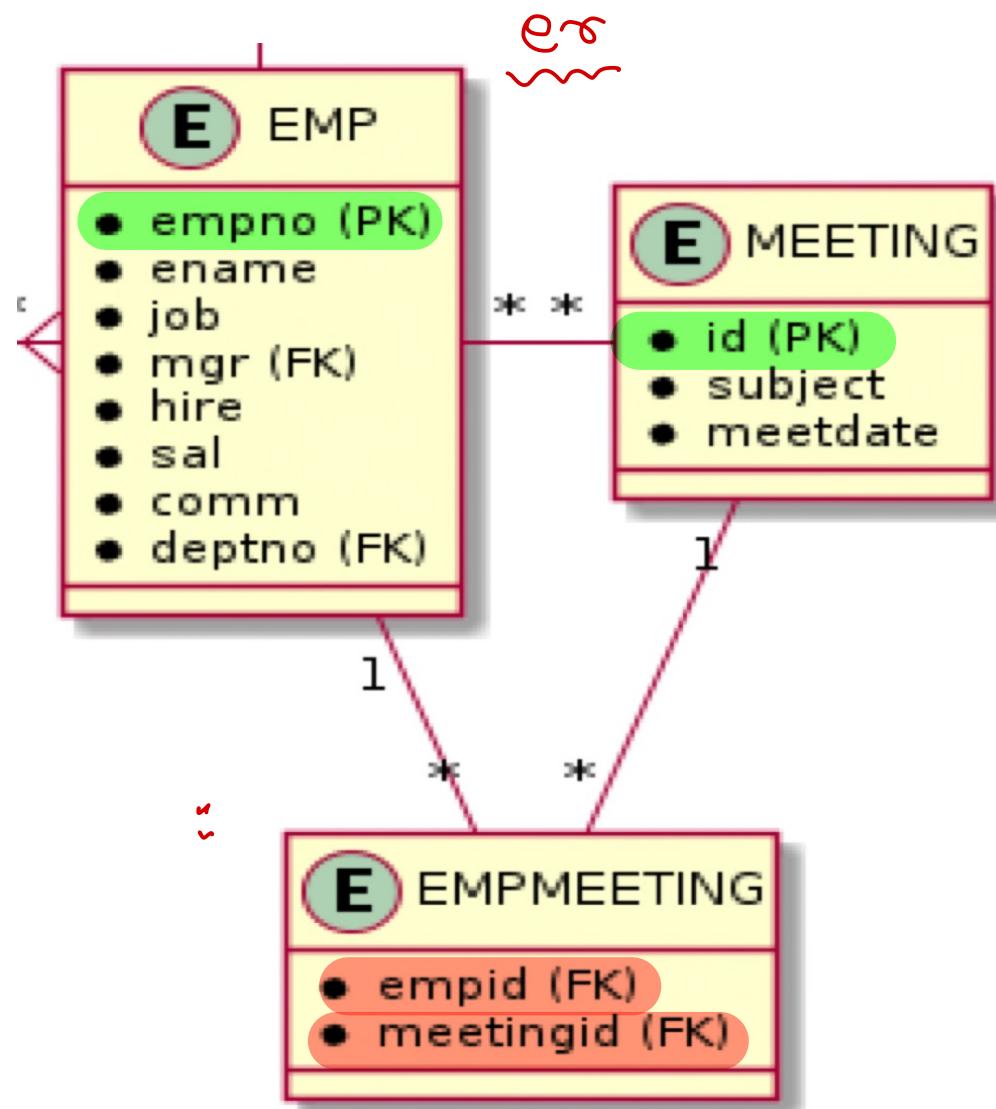
```
@ManyToOne
@JoinColumn(name="deptno")
private Dept dept;
```

# @ManyToMany (bi-directional)

- One Emp can have many Meetings.
- One Meeting will have many Emps.
- Many-to-many relation is established into two tables via an additional table (auxiliary table).
- The EMP\_MEETING table holds FK of both tables to establish the relation.
- In first class (e.g. Emp) use @ManyToMany along with @JoinTable (refering auxillary table & FK column in it).
  - joinColumn – first table's FK in aux table
  - inverseJoinColumn – second table's FK in aux table
- In second class (e.g. Meeting) use @ManyToMany with mappedBy to setup bi-directional relation.

```
class Emp {  
    @Id ✓  
    @Column private int empno; ✓  
    @Column private String ename; ✓  
    @ManyToMany  
    @JoinTable(name = "EMPMETING",  
    joinColumns = {@JoinColumn (name="EMPID")},  
    inverseJoinColumns = {@JoinColumn (name="MEETINGID")})  
    private List<Meeting> meetingList; ✓  
}  
  
class Meeting {  
    @Id ✓ PK  
    @Column private int id;  
    @Column private String subject; ✓  
    @ManyToMany(mappedBy="meetingList")  
    private List<Emp> empList;  
}
```

The diagram shows the relationship mapping between the Emp and Meeting classes. The Emp class has a many-to-many relationship with Meeting through the EMPMETING join table. The join table has foreign key columns EMPID and MEETINGID. The Meeting class has a many-to-many relationship with Emp through the same EMPMETING join table, using the mappedBy attribute to indicate the inverse side. The primary key (PK) for the Emp class is empno, and for the Meeting class is id.



## HQL Join

→ also used overriding fetch type given in orm annotations.  
from Emp e join Fetch e.meeting List where —

- SQL joins are used to select data from two related tables.
- Types of joins: Cross, Inner, Left outer, Right outer, Full outer.

### SQL Join syntax:

- SELECT e.ename, d.dname FROM EMP e INNER JOIN DEPT d ON e.deptno = d.deptno;
- ✓ SELECT e.ename, d.dname FROM EMP e, DEPT d WHERE e.deptno = d.deptno;

- HQL joins are little different than SQL joins. They can be used on association so that condition will be generated automatically. Alternatively property of composite object can be given in select or may use traditional join syntax.

### HQL Join syntax:

- SELECT e.ename, d.dname FROM EMP e INNER JOIN e.dept d;  
*entity name*
- SELECT e.ename, e.dept.dname FROM EMP e;  
*entity name*
- SELECT e.ename, d.dname FROM EMP e, DEPT d WHERE e.deptno = d.deptno;

} Object[]

- Note that joins are executed automatically when Hibernate associations are used.



# @OneToOne (bi-directional)

- One Emp have one Address.
- If both tables have same primary key, then use @OneToOne along along with @PrimaryKeyJoinColumn. Use @OneToMany with mappedBy in second class to setup bidirectional relation.
- If a table contains FK for another table use @OneToOne with @JoinColumn.

```
class Emp {  
    @Id  
    @Column private int empno;  
    }  
    {  
        @OneToOne  
        @JoinColumn(name="addr_id")  
        private Address addr;  
    }
```

```
class Address {  
    @Id  
    @Column private int id;  
    @Column private String country;  
    }  
    {  
        @OneToOne(mappedBy = "addr")  
        private Emp emp;  
    }
```

```
class Emp {  
    @Id  
    @Column private int empno;  
    }  
    {  
        @OneToOne  
        @PrimaryKeyJoinColumn  
        private Address addr;  
    }  
  
class Address {  
    @Id  
    @Column private int empid;  
    @Column private String country;  
    }  
    {  
        @OneToOne(mappedBy = "addr")  
        private Emp emp;  
    }
```

# Hibernate Fwd & Rev Engg



# Forward and Reverse Engineering

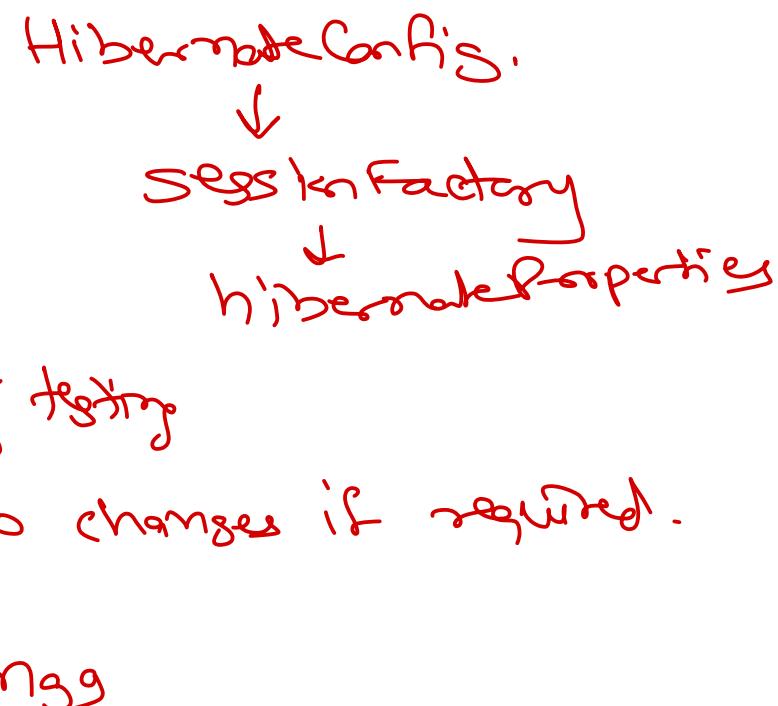
- There are two approaches
  - Database first approach (reverse engg)
  - Entity first approach (forward engg)
- Reverse engg
  - 1. Design database and create tables with appropriate relations. (PK + FK)
  - 2. Implement entity classes with appropriate ORM annotations
- Two ways to implement reverse engg
  - Manual approach
  - Wizard/tools (Eclipse -> JPA project)

JBoss → Hibernate Tools  
Plugin



# Forward and Reverse Engineering

- Forward engg
  - 1. Implement entity classes with appropriate ORM annotations.
  - 2. Database tables (with corresponding relations) will be automatically created when program is executed.
- Hibernate.cfg.xml – hibernate.hbm2ddl.auto
  - create: Db dropping will be generated followed by database creation.
  - create-only: Db creation will be generated.
  - create-drop: Drop the schema and recreate it on SessionFactory startup. Additionally, drop the schema on SessionFactory shutdown.
  - update: Update the database schema. → do changes if required.
  - validate: Validate the database schema
  - none: No action will be performed. → dev engg

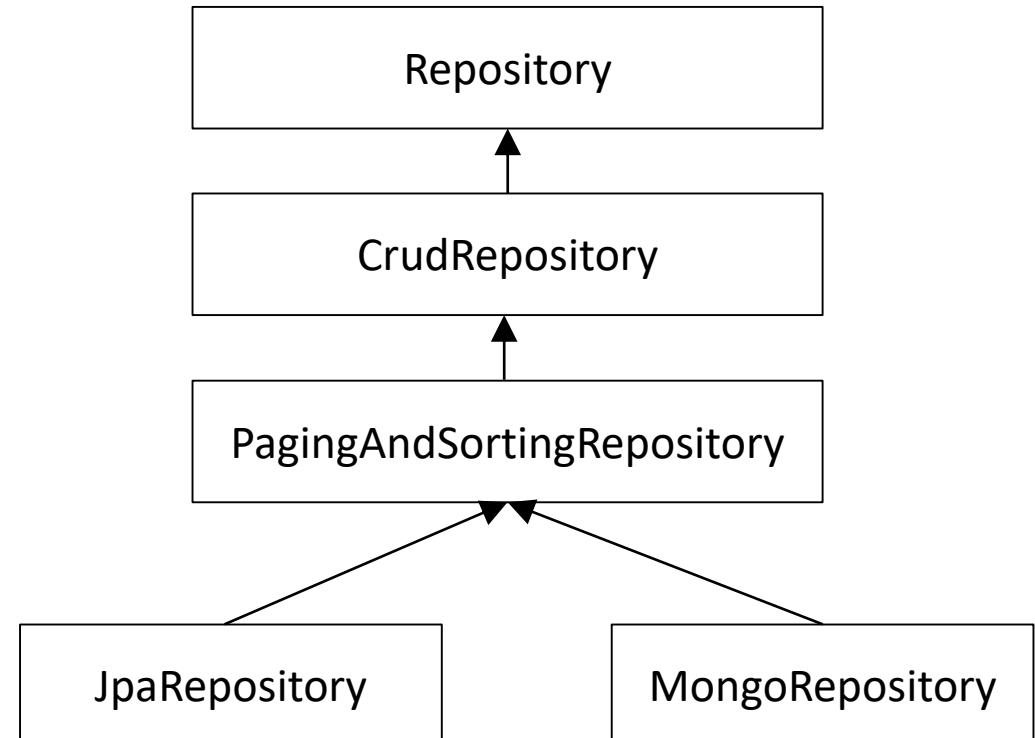


# Spring Data



# Spring Data

- Spring Data provides unified access to databases -- RDBMS or NoSQL.
- Spring Data JPA provides repository support for the Java Persistence API (JPA) -- RDBMS.
- Advantages
  - Eases development of applications that need to access JPA data sources.
  - Lot of boilerplate/repeated code is eliminated.
  - Consistent configuration and unified data access.



# Spring Data

- Available methods provide basic CRUD operations.
- For application specific database queries build queries using keywords.
  - Query expressions are usually property traversals combined with concatenation operators And / Or as well as comparison operators Between, LessThan, Like, etc.
  - IgnoreCase for individual properties or all properties.
  - OrderBy for static ordering.
- Query method examples
  - Customer findByName(String name);
  - List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
  - List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  - List<Person> findByLastnameIgnoreCase(String lastname);
  - List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  - Stream<Person> findByAgeBetween(int minAge, int maxAge);



# Spring Data

- Query method examples

- User findFirstByOrderByLastnameAsc();
- User findTopByOrderByAgeDesc();
- List<Person> findByAddressZipCode(ZipCode zipCode);
- Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);
- List<User> findFirst10ByLastname(String lastname, Sort sort);
- List<User> findTop10ByLastname(String lastname, Pageable pageable);

- Custom query methods

- Custom queries can be added using @Query annotation.

- JPQL (HQL) or SQL queries can be used

- `@Query("select distinct b.subject from Book b")`

```
List<String> findDistinctSubjects();
```

- `@Query(value="select distinct b.subject from BOOKS b", native=true)`

```
List<String> findDistinctSubjectsWithSQL();
```





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

# Java EE

## Agenda

- Hibernate entity relations
- Primary keys
- Stored procedure
- Criteria query
- Spring Data JPA

## Hibernate entity relations

### ManyToMany

- Refer slides for the concepts.
- Example:
  - emp table (empno (PK), ename, sal) --> @Entity class Emp
  - meeting table (id (PK), subject, meetdate) --> @Entity class Meeting
  - empmeeting table (empid (FK), meetingid (FK)) -- Auxillary tables
- Option 1: In Emp class @ManyToMany @JoinTable(...) and in Meeting class @ManyToMany(mappedBy="...")

```
// refer classroom code
```

- Option 2: In Meeting class @ManyToMany @JoinTable(...) and in Emp class @ManyToMany(mappedBy="...")

```
class Meeting {  
    @Id  
    private int id;  
    // ...  
    @ManyToMany  
    @JoinTable(table = "empmeeting",  
              joinColumns = {@JoinColumn(name="meetingid")},  
              inverseJoinColumns = {@JoinColumn(name="empid")})  
    private List<Emp> empList;  
}  
class Emp {  
    @Id  
    private int empno;  
    // ...  
    @ManyToMany(mappedBy="empList")  
    private List<Meeting> meetingList;  
}
```

## Demo 15 (ManyToMany)

- New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Data JPA
    - MySQL Driver
  - Finish
- Create database.properties file. Add database & hibernate properties in it.
- Create HibernateConfig class.
  - Mark it @Configuration, @EnableTransactionManagement and @PropertySource("classpath:database.properties")
  - Add fields to get values from properties file using @Value.
  - Create beans (@Bean) dataSource, sessionFactory and transactionManager.
- Create Emp and Meeting pojo class. Add appropriate ORM annotations.
- In Emp class add meetingList with @ManyToMany and @JoinTable annotation.
- Write a test Repository class TestRepository. Mark class with @Repository and @Transactional. @Autowired it into main class.
- Implement test method testGetEmpById() in TestRepository. Invoke method from main class. Observe SQL queries generated. Two queries are executed. Note that default fetch type is "lazy" for ManyToMany.
- In Emp class for meetingList change fetch type as "EAGER". Execute the application again. Observe SQL queries generated. Single JOIN query is executed. Change fetch type back to LAZY (before next step).
- In Meeting class add empList with @ManyToMany annotation with mappedBy="meetingList".
- Implement test method testGetMeetingById() in TestRepository. Invoke method from main class. Observe SQL queries generated. Two queries are executed. Note that default fetch type is "lazy" for ManyToMany.
- In Emp class for meetingList add cascade type as "MERGE". Add helper method addMeeting in Emp class. Also in Meeting class add helper method addEmp in Meeting class.
- Implement test method testAddMeeting() in TestRepository. Invoke method from main class. Observe SQL queries generated. Verify the entries in meeting and empmeeting table.
- Implement test method testGetEmpByName() in TestRepository. Use HQL query "from Emp e where e.ename=:p\_ename". Invoke method from main class. Observe SQL queries generated. Irrespective of fetch type, two queries are executed.
- In testGetEmpByName(), modify HQL query as "from Emp e join fetch e.meetingList where e.ename=:p\_ename". Execute the application again. Observe SQL queries generated. Irrespective of fetch type, Single JOIN query are executed.

## OneToOne

- Refer slides for the concepts.

## Demo 15 (OneToOne)

- Demo 15 is continued.
- Add entity class Address with appropriate ORM annotations.
- In Emp class add "Address addr" field with @OneToOne and @PrimaryKeyJoinColumn annotation.

- Implement test method `testGetEmpAddress()` in `TestRepository`. Invoke method from main class. Observe SQL queries generated. Irrespective of fetch type, two queries are executed back to back.

## Auto-generated Primary Key

- Refer slides for the concepts.

### Demo 16 (Auto generated Ids)

- New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Data JPA
    - MySQL Driver
  - Finish
- Create `database.properties` file. Add database & hibernate properties in it.
- Create `HibernateConfig` class.
  - Mark it `@Configuration`, `@EnableTransactionManagement` and `@PropertySource("classpath:database.properties")`
  - Add fields to get values from properties file using `@Value`.
  - Create beans (`@Bean`) `dataSource`, `sessionFactory` and `transactionManager`.
- Create `Customer` and `Book` pojo class. Add appropriate ORM annotations.
- In `Customer` class add `@GeneratedValue(strategy = IDENTITY)` on its id field.
- Write a test Repository class `TestRepository`. Mark class with `@Repository` and `@Transactional`. `@Autowired` it into main class.
- Implement test method `testAddCustomer()` in `TestRepository`. Invoke method from main class. Observe SQL queries generated. `INSERT` query is not having ID field, it is auto-generated in RDBMS due to `AUTO_INCREMENT` column.
- In `Book` class add `@GeneratedValue(strategy = IDENTITY)` on its id field.
- Implement test method `testAddBook()` in `TestRepository`. Invoke method from main class. The method fails, because books table id is not `AUTO_INCREMENT` in database.
- In database, create a table for book id.

```
CREATE TABLE bookid_gen(next_val INT);
INSERT INTO bookid_gen VALUES (70);
```

- In `Book` class change `@GeneratedValue` to `@GeneratedValue(generator = "bookid_gen", strategy = GenerationType.AUTO)` on id field.
- Run the application again. Observe SQL queries generated. Id is fetched from `bookid_gen` table, updated (increment) in database and used to insert Book.
- In `Book` class change `@GeneratedValue` for sequence generation

```
@SequenceGenerator(name = "gen", sequenceName = "bookid_gen", initialValue = 80)
@GeneratedValue(generator = "gen", strategy = GenerationType.SEQUENCE)
```

- Run the application again. Observe SQL queries generated. Id is fetched from bookid\_gen table, updated (increment) in database and used to insert Book.
- In database, create a table for bookshop\_ids.

```
CREATE TABLE bookshop_ids(entity CHAR(30) PRIMARY KEY, id INT);
```

- In Book class change @GeneratedValue for table based id generation

```
@TableGenerator(name = "gen", table = "bookshop_ids",
    pkColumnName = "entity",
    valueColumnName = "id",
    pkColumnValue = "book",
    initialValue = 300,
    allocationSize = 1)
@GeneratedValue(generator = "gen", strategy = GenerationType.TABLE)
```

- Run the application again. Observe SQL queries generated. A new book is inserted with new id. Note that INSERT is also done in bookshop\_ids. Change book details in testAddBook() and run application again. A new book is inserted and only update (not insert) is done on bookshop\_ids table.
- In Customer class change @GeneratedValue for table based id generation

```
@TableGenerator(name = "gen", table = "bookshop_ids",
    pkColumnName = "entity",
    valueColumnName = "id",
    pkColumnValue = "customer",
    initialValue = 1000,
    allocationSize = 1)
@GeneratedValue(generator = "gen", strategy = GenerationType.TABLE)
```

- Call testAddCustomer() from main class. Observe SQL queries generated. A new customer is inserted with new id. Note that INSERT is also done in bookshop\_ids.
- In database, observe contents of bookshop\_ids table. SELECT \* FROM bookshop\_ids.

### allocationSize

- Used with TableGenerator and SequenceGenerator.
- allocationSize means how many ids to be generated in a single call to the database.
- If allocationSize = 10, when hibernate access database to get the next id it is producing range of 10 ids.
- Example: If current id=400, then you are generating ids from 401 to 410. These ids can be used for inserting multiple entities (401, 402, 403, ...).
- Also hibernate is updating id = 410.
- If you try to add more than 10 entities, for 10 entities it will use the ids in range 401 to 410. For the next entity, it will select again from the database.

# Spring Data

- We will cover theory in the next lecture.

## Demo 17 (Spring Data)

- New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Data JPA
    - MySQL Driver
  - Finish
- In application.properties add database/hibernate properties.
- Create entity class Book (in sub-package of main class).
- Create BookDao interface (with no methods) inherited from JpaRepository.
- Implement BookDaoTest class. Mark it as @Component and @Transactional. Autowire BookDao in it.
- In BookDaoTest class add method testGetBook(). Call method from main(). Observe generated SQL query.
- In BookDaoTest class add method testAddBook(). Call method from main(). Observe generated SQL query.
- In BookDao interface add List findBySubject(String subject). (method naming convention must be strictly followed). In BookDaoTest add testGetBookBySubject(). Call method from main(). Observe generated SQL query.
- In BookDao interface add List findBySubjectAndAuthor(String subject, String author). (method naming convention must be strictly followed). In BookDaoTest add testGetBookBySubjectAndAuthor(). Call method from main(). Observe generated SQL query.



# Spring and Hibernate

*Nilesh Ghule <nilesh@sunbeaminfo.com>*



# JPA

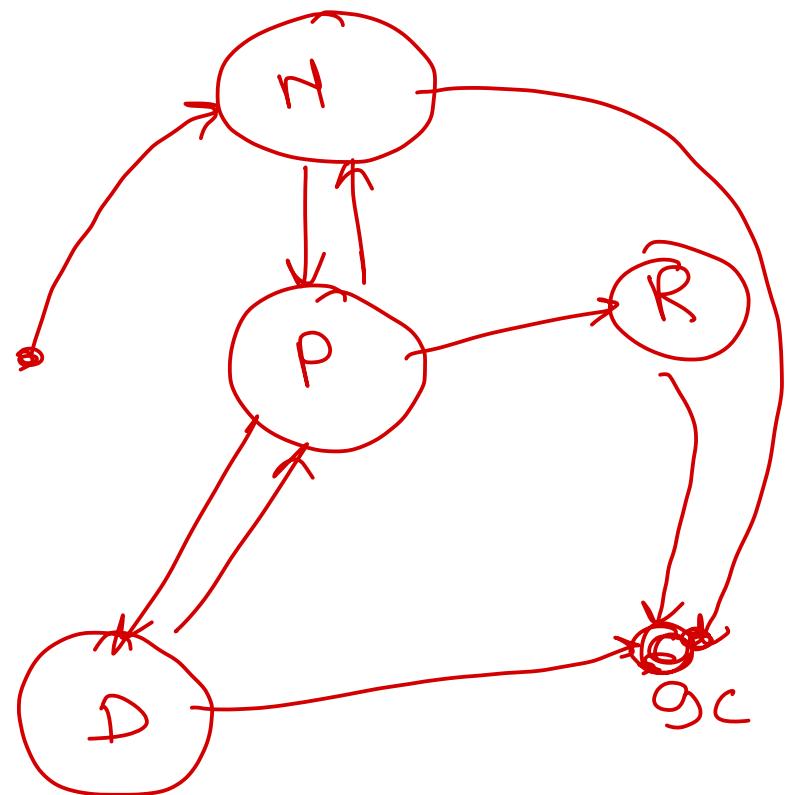


Sunbeam Infotech

[www.sunbeaminfo.com](http://www.sunbeaminfo.com)

# JPA

- JPA is specification for ORM.
- All ORM implementations follow JPA specification.
  - Hibernate, Torque, iBatis, EclipseLink, ...
- Hibernate implements JPA specs.
  - SessionFactory → EntityManagerFactory
  - Session → EntityManager
    - find(), persist(), merge(), refresh(), remove(), detach(), ...
  - HQL → JPQL
  - hibernate.cfg.xml → persistence.xml
- JPA versions (Standard)
  - 1.0, 1.1, 2.0, 2.1, 2.2
- JPA Entity life cycle
  - New (Transient)
  - Managed (Persistent)
  - Detached (Detached)
  - Removed (Removed)



# Spring JPA



# Spring JPA Integration

- Spring DI simplifies JPA.
- LocalEntityManagerFactoryBean bean provides entity manager factory, while transaction automation is done by JpaTransactionManager bean.
- Steps:
  - In pom.xml, add spring-orm, mysql-connector-java, hibernate-core.
  - Configure META-INF/persistence.xml.
  - Create entityManagerFactory (with JPA PersistenceUnitName configured), transactionManager beans. Also set default transactionManager.
  - Implement entity classes.
  - Implement @Repository class & auto-wire entity manager using @PersistentContext EntityManager em & then perform operations.
  - Implement @Service layer and mark business logic methods as @Transactional.
    - Note that single business operation (from service layer) may deal with multiple operations on different repositories. @Transactional put all ops under same tx.

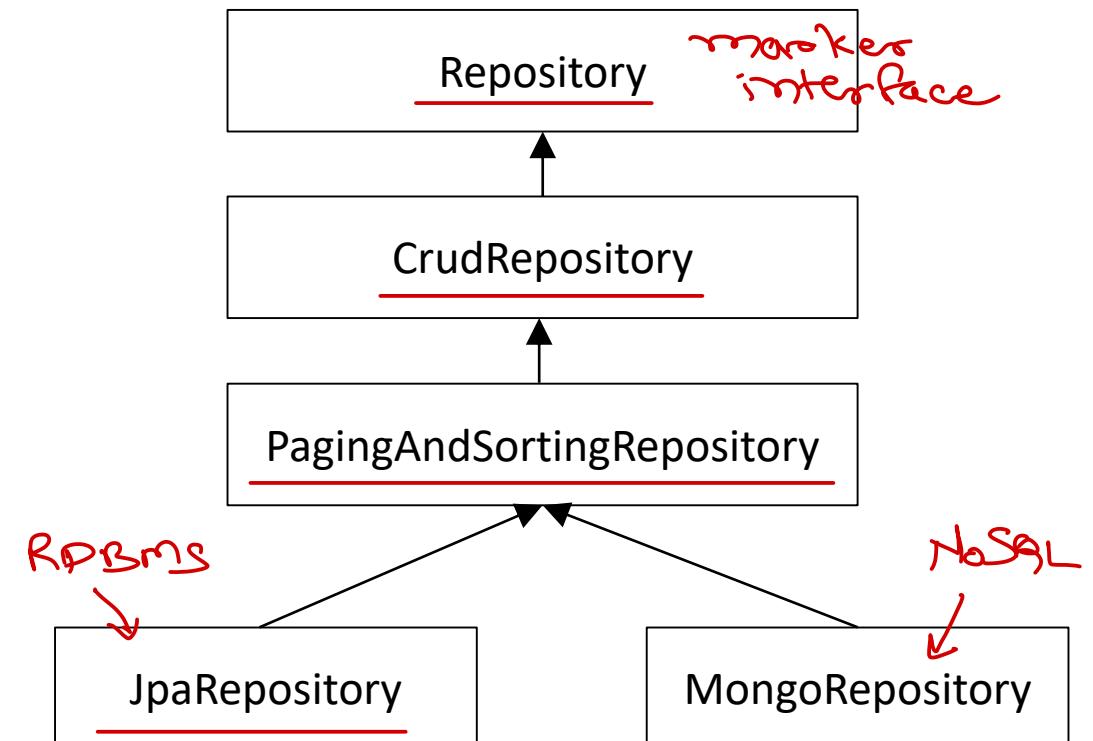
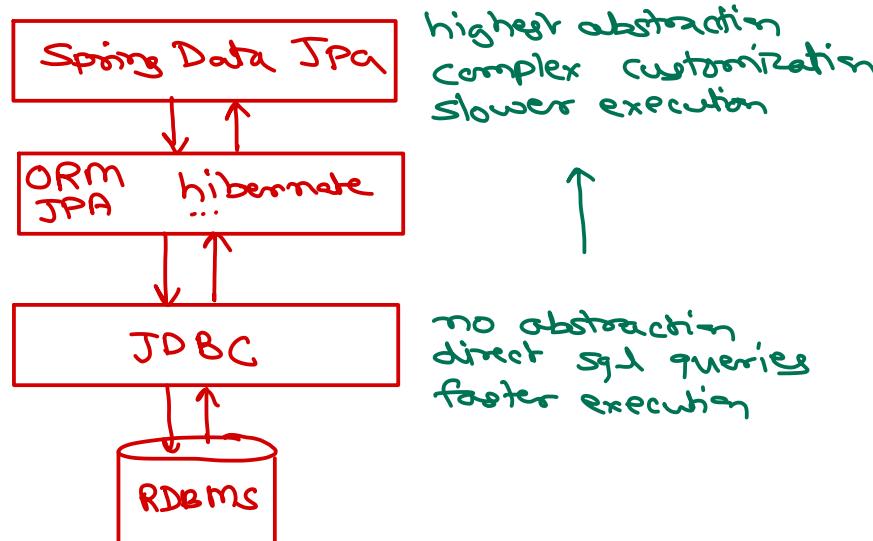


# Spring Data

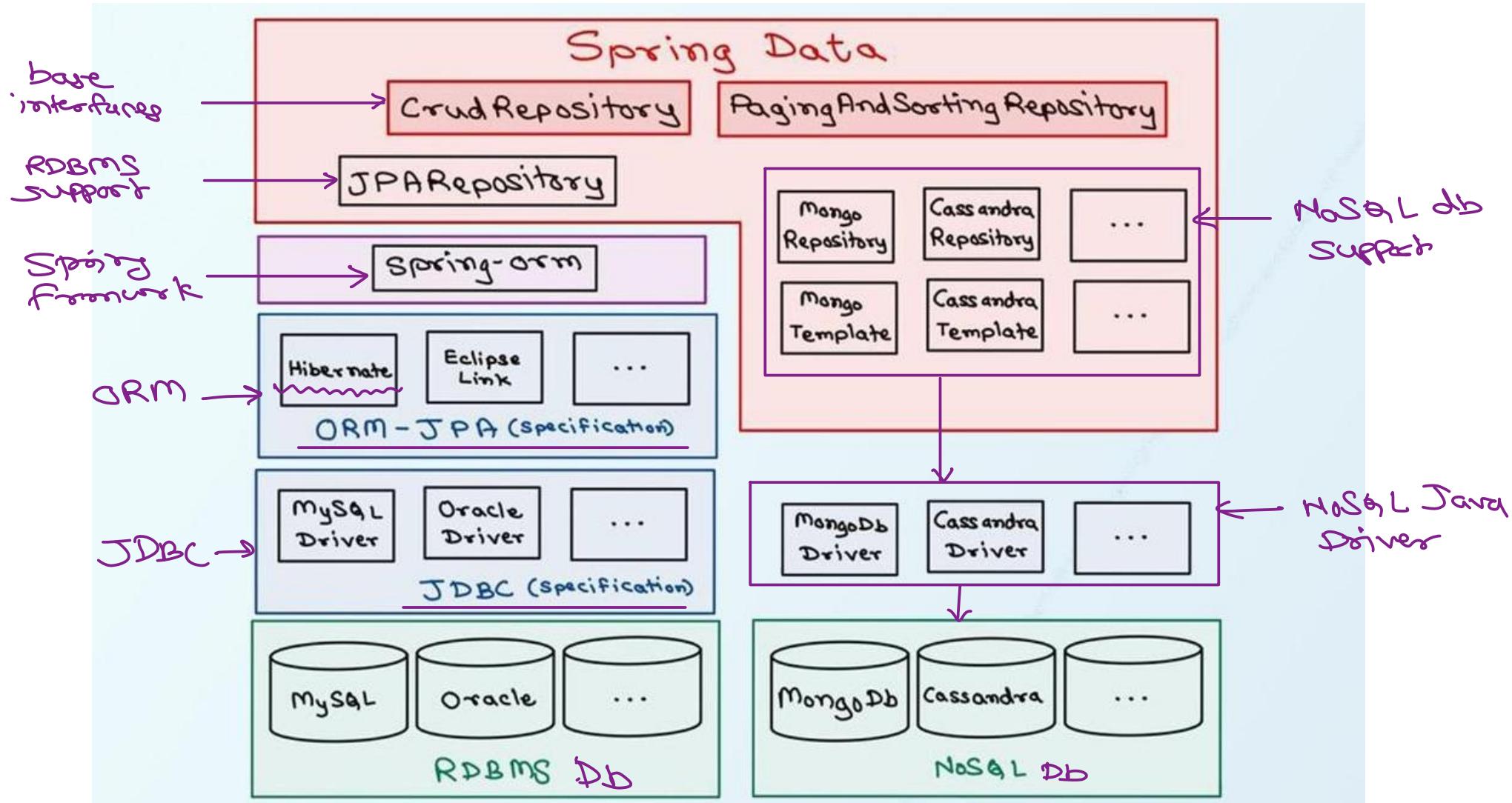


# Spring Data

- Spring Data provides unified access to databases -- RDBMS or NoSQL.
- Spring Data JPA provides repository support for the Java Persistence API (JPA) -- RDBMS.
- Advantages
  - Eases development of applications that need to access JPA data sources.
  - Lot of boilerplate/repeated code is eliminated.
  - Consistent configuration and unified data access.



# Spring Data



# Spring Data

Cloud Repository <>

- Available methods provide basic CRUD operations.
- For application specific database queries build queries using keywords.
  - Query expressions are usually property traversals combined with concatenation operators And / Or as well as comparison operators Between, LessThan, Like, etc.
  - IgnoreCase for individual properties or all properties.
  - OrderBy for static ordering.
- Query method examples
  - Customer Customer findByname(String name);
  - List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
  - List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  - List<Person> findByLastnameIgnoreCase(String lastname);
  - List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  - Stream<Person> findByAgeBetween(int minAge, int maxAge);

By  
And  
where

Properties in  
entity class



# Spring Data

- Query method examples

- User findFirstByOrderByLastnameAsc();
- User findTopByOrderByAgeDesc();
- List<Person> findByAddressZipCode(ZipCode zipCode);
- Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);
- List<User> findFirst10ByLastname(String lastname, Sort sort);
- List<User> findTop10ByLastname(String lastname, Pageable pageable);

- Custom query methods

- Custom queries can be added using @Query annotation.

- JPQL (HQL) or SQL queries can be used

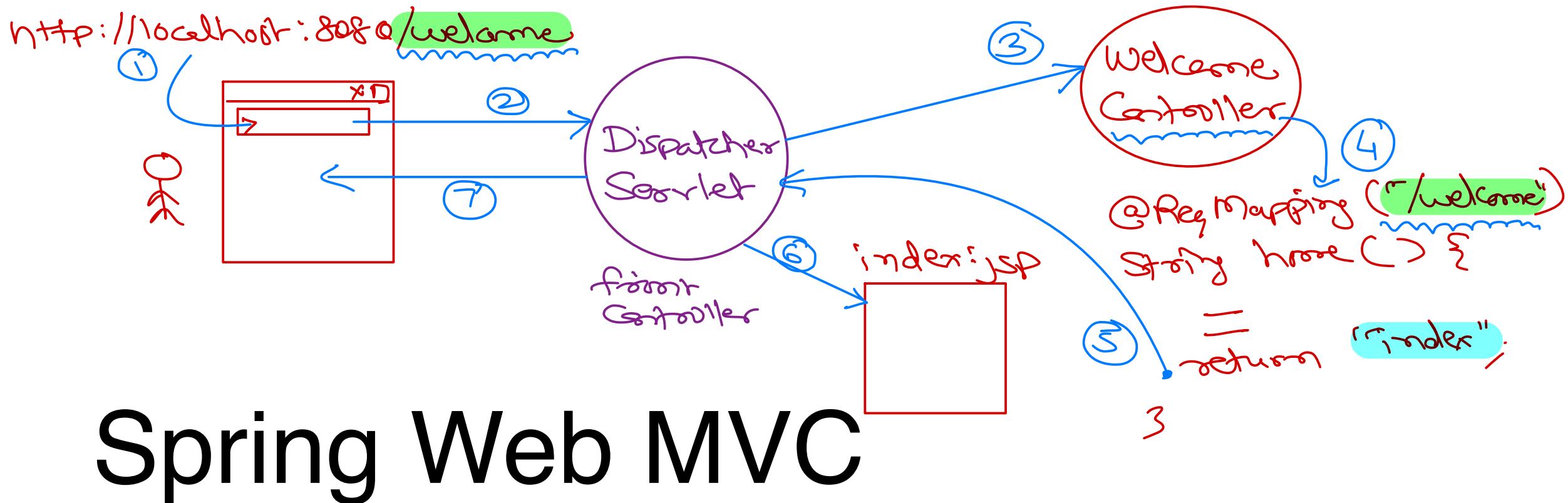
- `@Query("select distinct b.subject from Book b")`

```
List<String> findDistinctSubjects();
```

- `@Query(value="select distinct b.subject from BOOKS b", native=true)`

```
List<String> findDistinctSubjectsWithSQL();
```





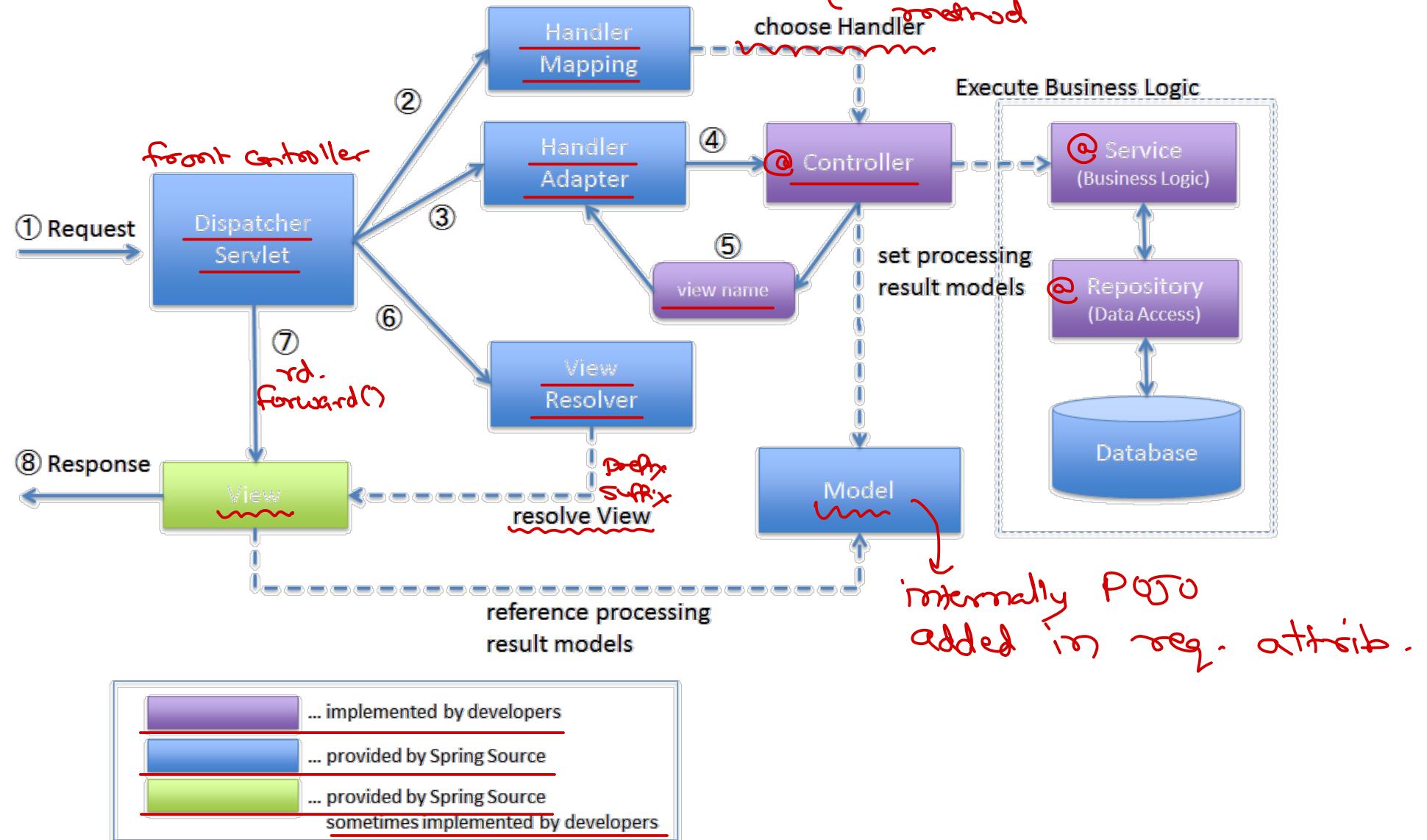
# Spring Web MVC

# Spring Web MVC

- MVC is design-pattern.
  - Divide application code into multiple relevant components to make application maintainable and extendable.
  - M: Model: Data of the application.
  - V: View: Appearance of data.
  - C: Controller: Interaction between models & views.
- Typical MVC implementation using Servlets & JSP.
  - Model: Java beans
  - View: JSP pages
  - Controller: Servlet dispatching requests
- Spring MVC components
  - Model: POJO classes holding data between view & controller.
  - View: JSP pages
  - Controller: Spring Front Controller i.e. DispatcherServlet
  - User defined controller: Interact with front controller to collect/send data to appropriate view, process with service layer.



# Spring Web MVC



# Spring Web MVC

- DispatcherServlet receives the request.
  - DispatcherServlet dispatches the task of selecting an appropriate controller to HandlerMapping. HandlerMapping selects the controller which is mapped to the incoming request URL and returns the (selected Handler) and Controller to DispatcherServlet.
  - DispatcherServlet dispatches the task of executing of business logic of Controller to HandlerAdapter.
  - HandlerAdapter calls the business logic process of Controller.
  - Controller executes the business logic, sets the processing result in Model and returns the logical name of view to HandlerAdapter.
  - DispatcherServlet dispatches the task of resolving the View corresponding to the View name to ViewResolver. ViewResolver returns the View mapped to View name.
  - DispatcherServlet dispatches the rendering process to returned View.
  - View renders Model data and returns the response.
- 
- <https://terasolunaorg.github.io/guideline/1.0.x/en/Overview/SpringMVCOversview.html>



# Spring Web MVC Annotation Config

- pom.xml: properties → <failOnMissingWebXml>false</failOnMissingWebXml>
- web.xml is replaced by MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer
  - Encapsulate declaration of dispatcher servlet.
  - Required for creating spring container/context.
    - getRootConfigClasses(): return config class for root webapplicationcontext.
    - getServletConfigClasses(): return config class for servlet webapplicationcontext
    - getServletMappings(): return dispatcher servlet url pattern (/)
- spring5-servlet.xml is replaced by MyWebMvcConfig implements WebMvcConfigurer
  - @Configuration
  - @ComponentScan(...): scans stereo-type annotations + other @Configuration classes.
  - @EnableWebMvc: creates AnnotationConfigWebApplicationContext.
  - @Bean: viewResolver



# Spring Boot Web MVC

- Create new Spring Boot project with Spring Web starter.
- In pom.xml add JSP & JSTL support
  - org.apache.tomcat.embed - tomcat-embed-jasper: provided
  - javax.servlet - jstl
- Configure viewResolver in application.properties
  - spring.mvc.view.prefix=/WEB-INF/jsp/
  - spring.mvc.view.suffix=.jsp
- Implement a controller with a request handler method returning view name (input).
- Under src/main create directory structure webapp/WEB-INF/jsp.
- Create input.jsp under webapp/WEB-INF/jsp to input user name and submit to server.
- Add another request handler method in controller to accept request param and send result to output page.
- Create output.jsp under webapp/WEB-INF/jsp to display the result.



# Request handler method → Inside @Controller class

- @RequestMapping attributes
  - value/path = “url-pattern”
  - method = GET | POST | PUT | DELETE
  - params/header = ... (map request only if given param or header is present).
  - consumes = ... (map request only if given request body type is available).
  - produces = ... (produce given response type from handler method)
- One request handler method can be mapped to multiple request methods.
- One request handler method can be restrict to set of request methods.
- To restrict handler method to single request method, shorthand annotations available.
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping



# Request handler method

- An @RequestMapping handler method can have a very flexible signatures.
- Supported method argument types
  - HttpServletRequest, HttpServletResponse
  - @RequestParam, @RequestHeader, @PathVariable
  - Map or Model or ModelMap → to send data to view.
  - Command object, @ModelAttribute, Errors or BindingResult
  - InputStream, OutputStream
  - HttpSession, @CookieValue, Locale
  - HttpEntity<>, @RequestBody.
- Supported method return types
  - String, View, Model or Map, ModelAndView
  - HttpHeaders, void
  - HttpEntity<>, ResponseEntity<>, @ResponseBody.





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

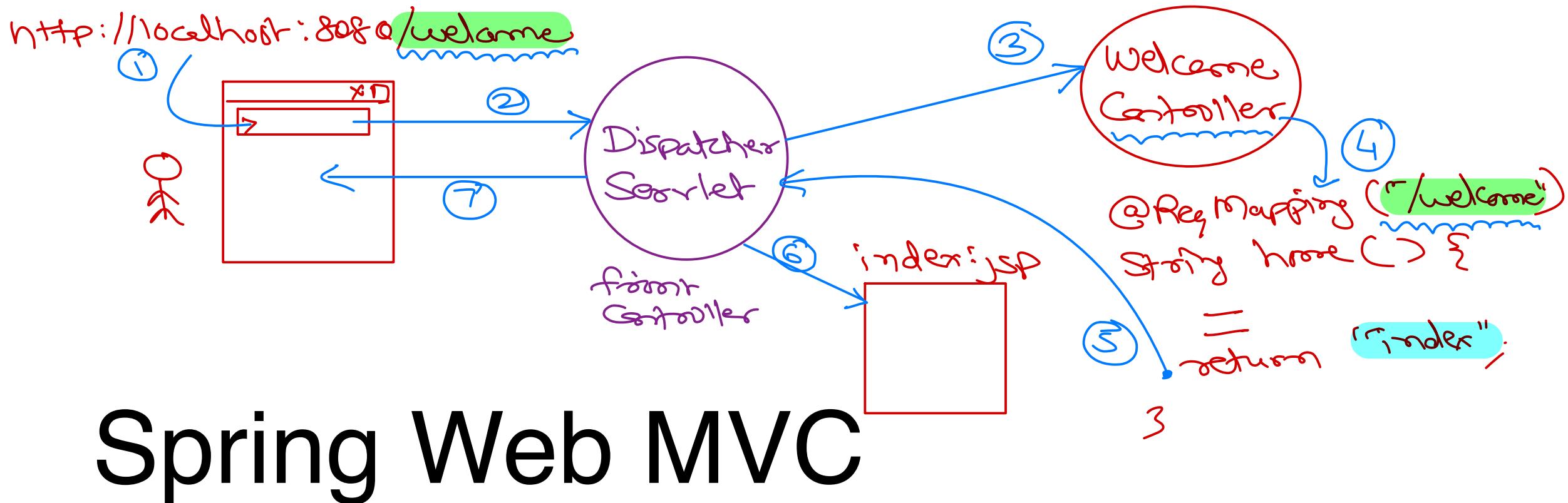




# Spring and Hibernate

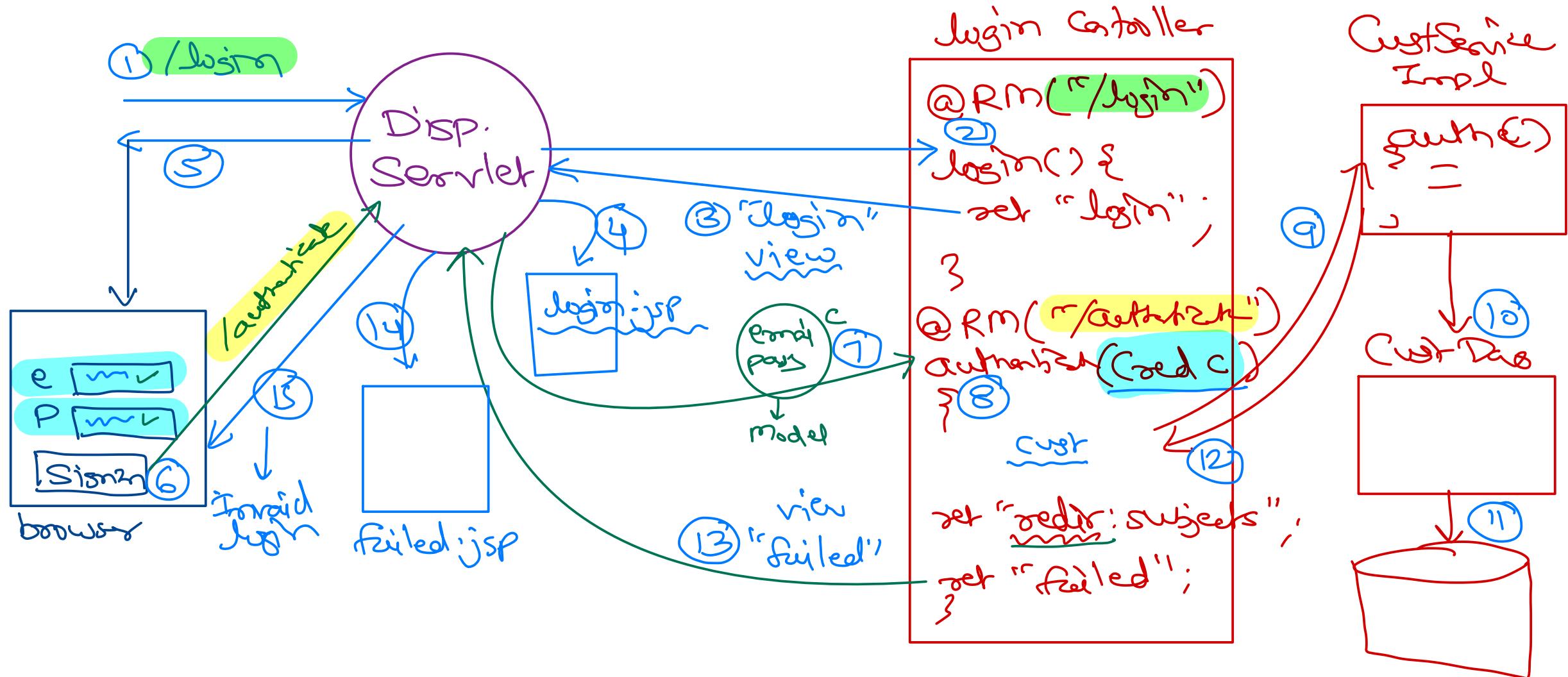
*Nilesh Ghule <nilesh@sunbeaminfo.com>*



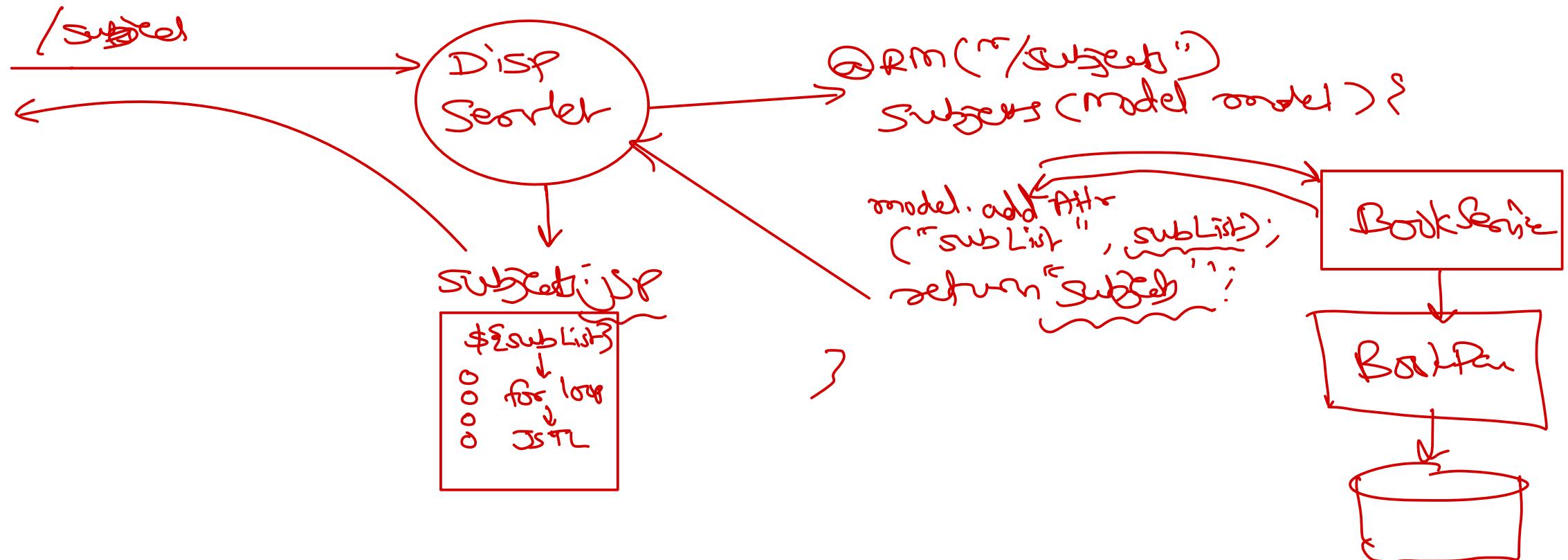


# Spring Web MVC

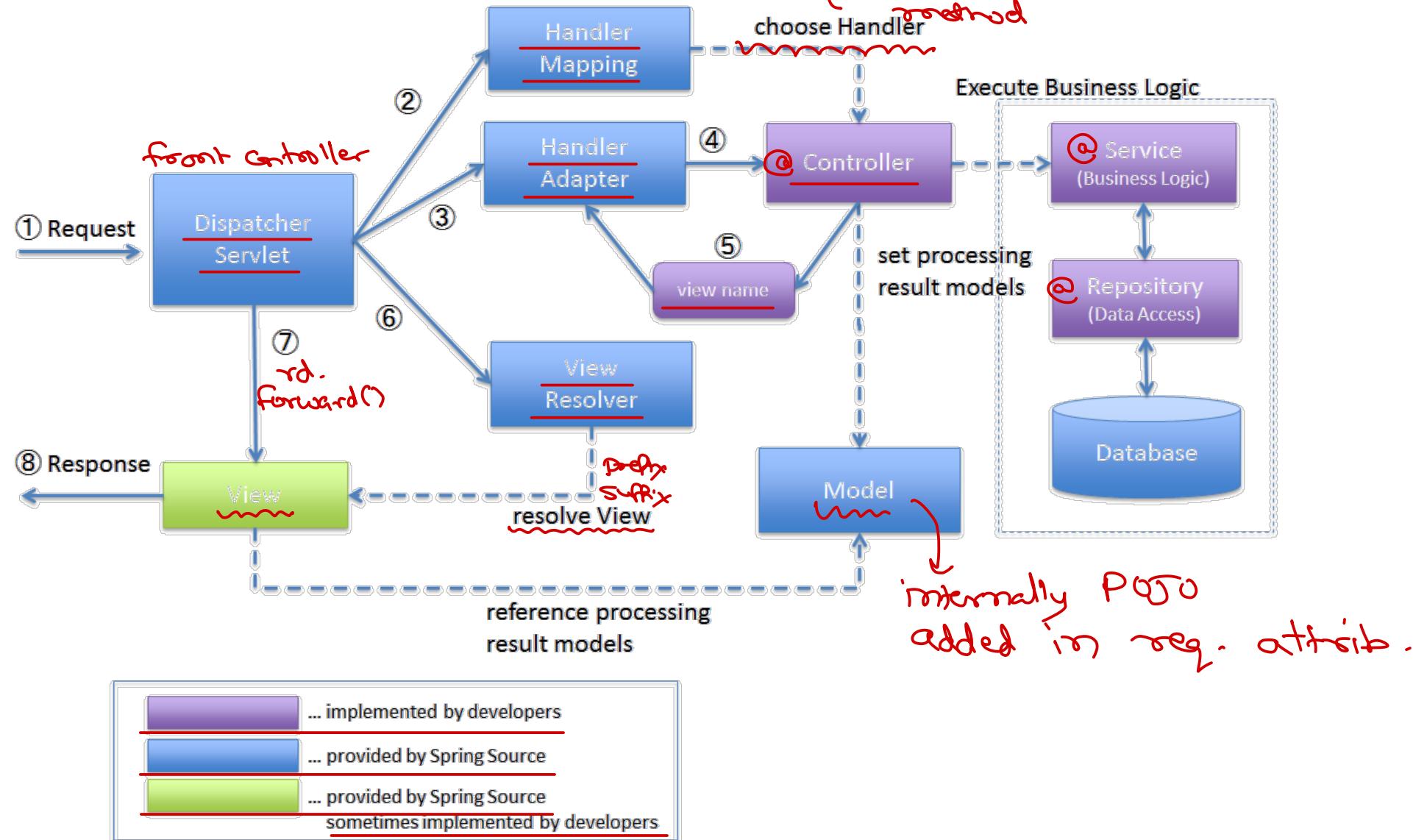
# Spring Web MVC



# Spring Web MVC



# Spring Web MVC





**Thank you!**

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>



# Java EE

## Agenda

- Spring Data
- JPA
- Spring MVC

## Spring Data

- "Spring Data" provides unified access to databases -- RDBMS or NoSQL.
- "Spring Data JPA" provides repository support for the Java Persistence API (JPA) -- RDBMS.

### Spring Data Architecture

#### Data JPA interfaces

- Repository -- Marker interface

```
public interface Repository<T, ID> {  
    // empty  
}
```

- CrudRepository

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {  
    long count()  
    // Returns the number of entities available.  
    void delete(T entity)  
    // Deletes a given entity.  
    void deleteAll()  
    // Deletes all entities managed by the repository.  
    void deleteAll(Iterable<? extends T> entities)  
    // Deletes the given entities.  
    void deleteById(ID id)  
    // Deletes the entity with the given id.  
    boolean existsById(ID id)  
    // Returns whether an entity with the given id exists.  
    Iterable<T> findAll()  
    // Returns all instances of the type.  
    Iterable<T> findAllById(Iterable<ID> ids)  
    // Returns all instances of the type with the given IDs.  
    Optional<T> findById(ID id)  
    // Retrieves an entity by its id.  
    <S extends T> S save(S entity)  
    // Saves a given entity.  
    <S extends T> Iterable<S> saveAll(Iterable<S> entities)
```

```
//Saves all given entities.
}
```

- PagingAndSortingRepository

```
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {
    Iterable<T> findAll(Sort sort);
    // return object's collection as per sort order
    Page<T> findAll(Pageable pageable);
    // return object's collection in a Page<> as per paging spec.
}
```

```
// Example: Fetch second page (index=1) of size 20
Page<User> users = dao.findAll(PageRequest.of(1, 20));
```

- JpaRepository

- Inherited from PagingAndSortingRepository, CrudRepository, QueryByExampleExecutor.

```
public interface JpaRepository<T, ID> extends CrudRepository<T, ID>, ... {
    void deleteAllInBatch();
    // Deletes all entities in a batch call.
    void deleteInBatch(Iterable<T> entities);
    // Deletes the given entities in a batch which means it will create a
    single Query.
    <S extends T> List<S> findAll(Example<S> example);
    <S extends T> List<S> findAll(Example<S> example, Sort sort);
    List<T> findAllById(Iterable<ID> ids);
    void flush();
    // Flushes all pending changes to the database.
    T getOne(ID id);
    // Returns a reference to the entity with the given identifier.
}
```

- MongoRepository

- Inherited from PagingAndSortingRepository, CrudRepository, QueryByExampleExecutor.

```
public interface MongoRepository<T, ID> extends CrudRepository<T, ID>, ... {
    List<T> findAll();
    List<T> findAll(Sort sort);
    <S extends T> S insert(S entity);
    ...
}
```

## Repository implementation

- The "implementation" of the Spring Data repository hidden (not to implement in code).
- Implemented by "SimpleJpaRepository"
  - Provide method implementations.
  - Handles transactions using @Transactional
- @Transactional is not necessary.
- To deal with multiple DAOs use @Transactional on @Service layer.

## Query Methods

- Available methods provide basic CRUD operations.
- For application specific database queries build queries using keywords.
  - Query expressions are usually property traversals combined with concatenation operators And / Or as well as comparison operators Between, LessThan, Like, etc.
  - IgnoreCase for individual properties or all properties.
  - OrderBy for static ordering.

## Query examples

```
Customer findByNome(String nome);
List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String
lastname);
// Enables the distinct flag for the query
List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String
firstname);
// Enabling ignoring case for an individual property
List<Person> findByLastnameIgnoreCase(String lastname);
// Enabling static ORDER BY for a query
List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
List<Person> findByAddressZipCode(ZipCode zipCode);
// Assuming a Person has an Address with a ZipCode. In that case, the method
creates the property traversal x.address.zipCode.
Stream<Person> findByAgeBetween(int minAge, int maxAge);
// Limiting the result size of a query with Top or First
User findFirstByOrderByLastnameAsc();
User findTopByOrderByAgeDesc();
Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);
Slice<User> findTop3ByLastname(String lastname, Pageable pageable);
List<User> findFirst10ByLastname(String lastname, Sort sort);
List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

## Query name parsing logic

- List findByAddressZipCode(ZipCode zipCode);
  - (1) from Person p where p.addressZipCode = :zipCode

```
class Person {
    private String addressZipCode;
    // ...
}
```

- (2) from Person p where p.addressZip.code = :zipCode

```
class AddressZip {
    int code;
    // ...
}
class Person {
    private AddressZip addressZip;
    // ...
}
```

- (3) from Person p where p.address.zipCode = :zipCode

```
class Address {
    int zipCode;
    // ...
}
class Person {
    private Address address;
    // ...
}
```

- List findByAddress\_ZipCode(ZipCode zipCode);
  - Manual traverse point: from Person p where p.address.zipCode = :zipCode

```
class Address {
    int zipCode;
    // ...
}
class Person {
    private Address address;
    // ...
}
```

## Query expression keywords

- Refer method subject keywords, predicate keywords and sorting keywords
  - <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods>

- <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#appendix.query.method.subject>
- <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#appendix.query.method.predicate>

## Custom query methods

- Custom queries can be added using @Query annotation.
- JPQL (HQL) queries
  - from Book b
  - from Book b order by b.price desc
  - from Book b where b.subject = :p\_subject
  - select distinct b.subject from Book b

```
@Query("select distinct b.subject from Book b")
List<String> findDistinctSubjects();
// JPA query targetting entities

@Query(value="select distinct b.subject from BOOKS b", native=true)
List<String> findDistinctSubjectsWithSQL();
// SQL query targetting RDBMS tables
```

## Calling Stored Procedures

- Spring data supports calling stored procedure from database using @Procedure annotation.
- Procedure returning one/more out parameters are supported by Spring data @Procedure.
- @Procedure mapped to procedure in database

```
DELIMITER $$

CREATE PROCEDURE findPriceById(IN bId INT, OUT bPrice DOUBLE)
BEGIN
    SELECT price INTO bPrice FROM books WHERE id=bId;
END;
$$

DELIMITER ;
```

- If no attributes are given, query method name is mapped to stored procedure name in database.

```
@Procedure
Double findPriceById(int id);
// CALL findPriceById(id, @out)
```

- value="procedureNameInDatabase" attribute mapped to stored procedure name in database.

```
@Procedure("findPriceById")
Double getPriceById(int id);
// CALL findPriceById(id, @out)
```

- procedureName="procedureNameInDatabase" attribute mapped to stored procedure name in database.

```
@Procedure(procedureName = "findPriceById")
Double callPriceById(int id);
// CALL findPriceById(id, @out)
```

- Note that output parameter is returned as method return type.
- @Procedure mapped to @NamedStoredProcedureQuery on entity class implicitly.

```
// If using JPA/Hibernate directly.
@NamedStoredProcedureQuery(name = "Book.nameById", procedureName =
"findNameById", parameters = {
    @StoredProcedureParameter(name = "bId", mode = ParameterMode.IN, type =
Integer.class),
    @StoredProcedureParameter(name = "bName", mode = ParameterMode.OUT, type =
String.class)
})
public class Book {
    // ...
}
```

```
// Using Spring Data -- @Param is not compulsory.
@Procedure
String nameById(@Param("bId") int id);
```

## JPA

- JPA is a specification for ORM tools.

### Spring Data Jpa version

- Spring Data JPA 1.10
  - Hibernate 5 (ORM - default)
  - OpenJPA 2.4 (ORM)

- EclipseLink 2.6.1 (ORM)
- QueryDSL 4 (Annotation processing)

## Spring Web MVC

- MVC is design-pattern.
- Divide application code into multiple relevant components to make application maintainable and extendable.
  - M: Model: Data of the application.
  - V: View: Appearance of data.
  - C: Controller: Interaction between business logic & views and Navigation.
- Custom MVC implementation using servlets and jsp:
  - Model: Java beans
  - View: JSP pages
  - Controller: ControllerServlet (typically -- load-on-startup=1, config)
- Spring MVC components
  - Model: POJO classes holding data between view & controller.
  - View: "JSP" or Thymeleaf or Freemarker pages
  - Controller: Spring Front Controller i.e. DispatcherServlet
  - User defined controller: Interact with front controller to collect/send data to appropriate view, process with service layer.

## Spring Web MVC Flow

- DispatcherServlet receives the request.
- DispatcherServlet dispatches the task of selecting an appropriate controller to HandlerMapping. HandlerMapping selects the controller which is mapped to the incoming request URL and returns the (selected Handler) and Controller to DispatcherServlet.
- DispatcherServlet dispatches the task of executing of business logic of Controller to HandlerAdapter.
- HandlerAdapter calls the request handler method of @Controller.
- Controller executes the business logic, sets the processing result in Model and returns the logical name of view to HandlerAdapter.
- DispatcherServlet dispatches the task of resolving the View corresponding to the View name to ViewResolver. ViewResolver returns the View Path mapped to View name.
- DispatcherServlet dispatches the rendering process to View class / View Engine.
- It renders view with Model data and returns the response.

## Request handler Methods

- @RequestMapping attributes
  - value/path = "url-pattern"
  - method = GET | POST | PUT | DELETE
  - params/header = ... (map request only if given param or header is present).
  - consumes = ... (map request only if given request body type is available).
  - produces = ... (produce given response type from handler method)
- One request handler method can be mapped to multiple HTTP request methods (get, post, ...).
- One request handler method can be restrict to set of HTTP request methods.
- To restrict handler method to single request method, shorthand annotations available.

- @GetMapping --> @RequestMapping(method="GET")
- @PostMapping --> @RequestMapping(method="POST")
- @PutMapping --> @RequestMapping(method="PUT")
- @DeleteMapping --> @RequestMapping(method="DELETE")

## Flexible signatures

- An @RequestMapping handler method can have a very flexible signatures.
- Supported method argument types
  - HttpServletRequest, HttpServletResponse
  - @RequestParam, @RequestHeader, @PathVariable
  - Map or Model or ModelMap
  - Command (Model) object, @ModelAttribute, Errors or BindingResult
  - InputStream, OutputStream
  - HttpSession, @CookieValue("cookie-name") String value, Locale
  - HttpEntity<>, @RequestBody -- for REST services.
- Supported method return types
  - String (viewName), View, Model or Map, ModelAndView
  - Command (Model) object
  - HttpHeaders, void - OutputStream
  - HttpEntity<>, ResponseEntity<>, @ResponseBody -- for REST services.

## Demo 19 (Simple MVC)

- New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Data JPA
    - MySQL Driver
    - Spring Web
  - Finish
- In pom.xml add dependencies for JSP & JSTL.

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>
```

- In application.properties add following properties

```
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
```

- Create class com.sunbeam.controllers.WelcomeController class.

```
@Controller
public class WelcomeController {
    @RequestMapping("/welcome") // like servlet url-pattern
    public String home(Model model) {
        model.addAttribute("curDate", new Date());
        return "index";           // view name = index --> /WEB-
INF/jsp/index.jsp
    }
}
```

- Create folder structure for JSP pages "webapp/WEB-INF/jsp" under src/main.
- Create webapp/WEB-INF/jsp/index.jsp.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Home</title>
</head>
<body>
    <h1>Welcome to Spring MVC</h1>
    <h3>Sunbeam Infotech</h3>
    <p>
        ${curDate}
    </p>
</body>
</html>
```

- Run as Spring Boot Application.
- Browser: http://localhost:8080/welcome

## Demo bookshop4

- New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Web
  - Finish
- In pom.xml add dependencies for JSP & JSTL.

```

<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>

```

- In application.properties add following properties

```

spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp

```

- Create com.sunbeam.controllers.LoginControllerImpl class with following method

```

@RequestMapping("/login") // url-pattern
public String login() {
    return "login"; // view-name: /WEB-INF/jsp/login.jsp
}

```

- Create folder structure for JSP pages "webapp/WEB-INF/jsp" under src/main.
- Create webapp/WEB-INF/jsp/login.jsp.

```

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Login</title>
</head>
<body>
    <h2>Online Book Shop</h2>
    <form method="post" action="authenticate">
        <table>
            <tr>
                <td>Email:</td>
                <td><input type="text" name="email"/></td>
            </tr>
            <tr>
                <td>Password:</td>
                <td><input type="password" name="password"/></td>
            </tr>
            <tr>
                <td colspan="2">
                    <input type="submit" value="Sign In"/>
                </td>
            </tr>
        </table>
    </form>
</body>

```

```

        <a href="#">Sign Up</a>
    </td>
</tr>
</table>
</form>
</body>
</html>

```

- Run application (Stop previously spring mvc running application, if any).
  - Browser: <http://localhost:8080/login>
- In application.properties file add properties (do changes for your database).

```

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
spring.datasource.url=jdbc:mysql://localhost:3306/sunbeam_db
spring.datasource.username=sunbeam
spring.datasource.password=sunbeam

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=none

```

- Create entity class com.sunbeam.entities.Customer with appropriate ORM annotations.
- Create dao interface com.sunbeam.daos.CustomerDao with appropriate methods.
- Create service interface com.sunbeam.services.CustomerService with required methods.
- Implement service interface com.sunbeam.services.CustomerServiceImpl with @Service and @Transactional annotations.
- Create model class com.sunbeam.models.Credentials to collect email and password from login.jsp.
- In LoginControllerImpl class add following method. Note that login.jsp form action="authenticate" is mapped to @RequestMapping("/authenticate"). Also method return to "failed" JSP page if login is invalid; otherwise will be redirected to the URL subjects (Not implemented so will get 404 now).

```

@RequestMapping("/authenticate")
public String authenticate(Credentials cred) { // Credentials object is
model obj to collect req param email and password.
    Customer cust = custService.authenticate(cred.getEmail(),
    cred.getPassword());
    if(cust != null)
        return "redirect:subjects"; // redirect to url subjects
    return "failed"; // view-name: /WEB-INF/jsp/failed.jsp
}

```

- Create webapp/WEB-INF/jsp/failed.jsp to display invalid login message.

## Assignment (Signup)

- Bookshop4 demo continued.
- Create JSP page webapp/WEB-INF/jsp/register.jsp for registration. Make form action="signup".

- In LoginControllerImpl class add following method.

```
@RequestMapping("/register") // url-pattern
public String register() {
    return "register"; // view-name
}
```

- Connect "register" URL pattern in Signup link in login.jsp.
- Run application and check if register page is launched on Sign Up link click.
- In com.sunbeam.entities.Customer class, add annotation on birthDate.

```
@DateTimeFormat(pattern="yyyy-MM-dd")
@Temporal(TemporalType.DATE)
private Date birthDate;
```

- In LoginControllerImpl class add following method.

```
@RequestMapping("/signup")
public String signup(Customer cust) {
    // call custService save() with try-catch.
    // if successful redirect to login.
    // if failed, go to view register.
}
```

- Run application and check if Sign Up functionality is working.

## bookshop4 (Subjects & Books)

- Bookshop4 demo continued.
- Create Book, BookDao, BookService and BookServiceImpl (similar to Customer classes), if not created already.
- Create com.sunbeam.controllers.BookControllerImpl class with subjects() method.
  - @RequestMapping("/subjects") <-- mapped to "redirect:subjects" in LoginControllerImpl.authenticate().
  - Get distinct subjects using BookService.
  - Add subjectList into model.
  - Return "subjects" <-- /WEB-INF/jsp/subjects.jsp view.
- Create /WEB-INF/jsp/subjects.jsp to display subjects radio buttons from "subjectList" added into model.
- Run application and check if Subjects functionality is working.
- Create com.sunbeam.controllers.BookControllerImpl class with books() method.
  - @RequestMapping("/books") <-- mapped to form action="books" in subjects.jsp.
  - Get selected subject radio button as a @RequestParam.
  - Get books of selected subject using BookService.
  - Add bookList into model.

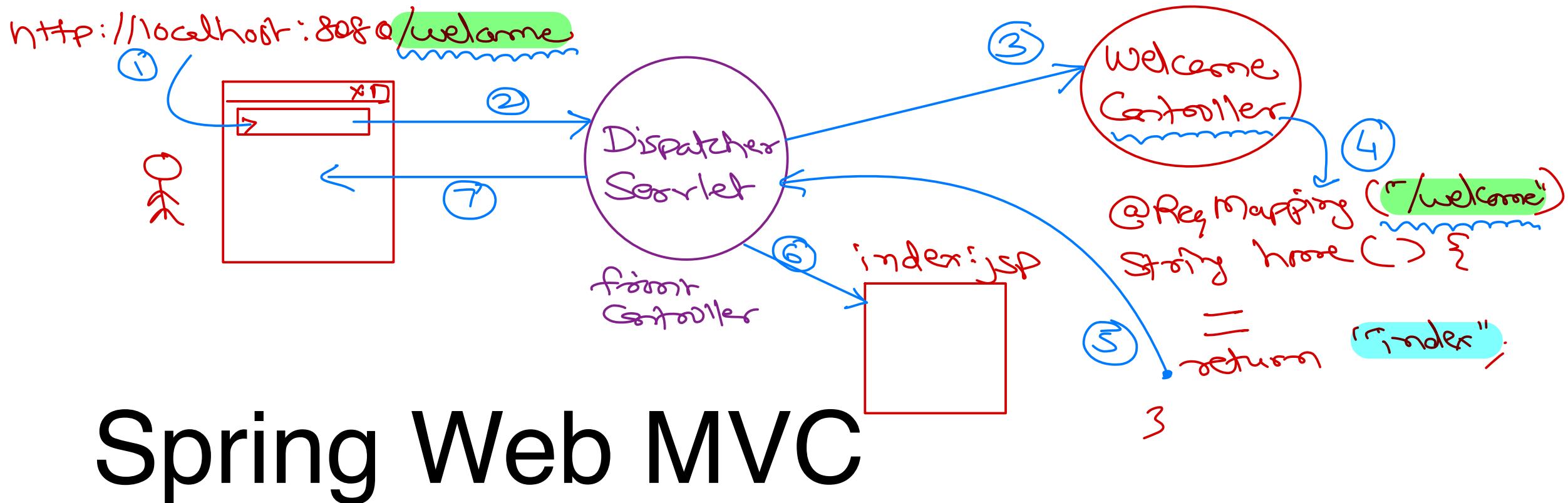
- Return "books" <-- /WEB-INF/jsp/books.jsp view.
- Create /WEB-INF/jsp/books.jsp to display books checkboxes from "bookList" added into model.



# Spring and Hibernate

*Nilesh Ghule <nilesh@sunbeaminfo.com>*





# Spring Web MVC

# Spring Web MVC Annotation Config (not using Spring boot) → maven pom

war → maven

- pom.xml: properties → <failOnMissingWebXml>false</failOnMissingWebXml>
- web.xml is replaced by MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer ← given by Spring mvc
  - Encapsulate declaration of dispatcher servlet.
  - Required for creating spring container/context.
    - getRootConfigClasses(): return config class for root webapplicationcontext.
    - getServletConfigClasses(): return config class for servlet webapplicationcontext
    - getServletMappings(): return dispatcher servlet url pattern (/)
- spring5-servlet.xml is replaced by MyWebMvcConfig implements WebMvcConfigurer
  - @Configuration
  - @ComponentScan(...): scans stereo-type annotations + other @Configuration classes.
  - @EnableWebMvc: creates AnnotationConfigWebApplicationContext.
  - @Bean: viewResolver → UrlBasedViewResolver bean  
- prefix , suffix

@Controller  
views



# Spring Boot Web MVC

- Create new Spring Boot project with Spring Web starter.
- In pom.xml add JSP & JSTL support
  - org.apache.tomcat.embed - tomcat-embed-jasper: provided
  - javax.servlet - jstl
- Configure viewResolver in application.properties
  - spring.mvc.view.prefix=/WEB-INF/jsp/
  - spring.mvc.view.suffix=.jsp
- Implement a controller with a request handler method returning view name (input).
- Under src/main create directory structure webapp/WEB-INF/jsp.
- Create input.jsp under webapp/WEB-INF/jsp to input user name and submit to server.
- Add another request handler method in controller to accept request param and send result to output page.
- Create output.jsp under webapp/WEB-INF/jsp to display the result.



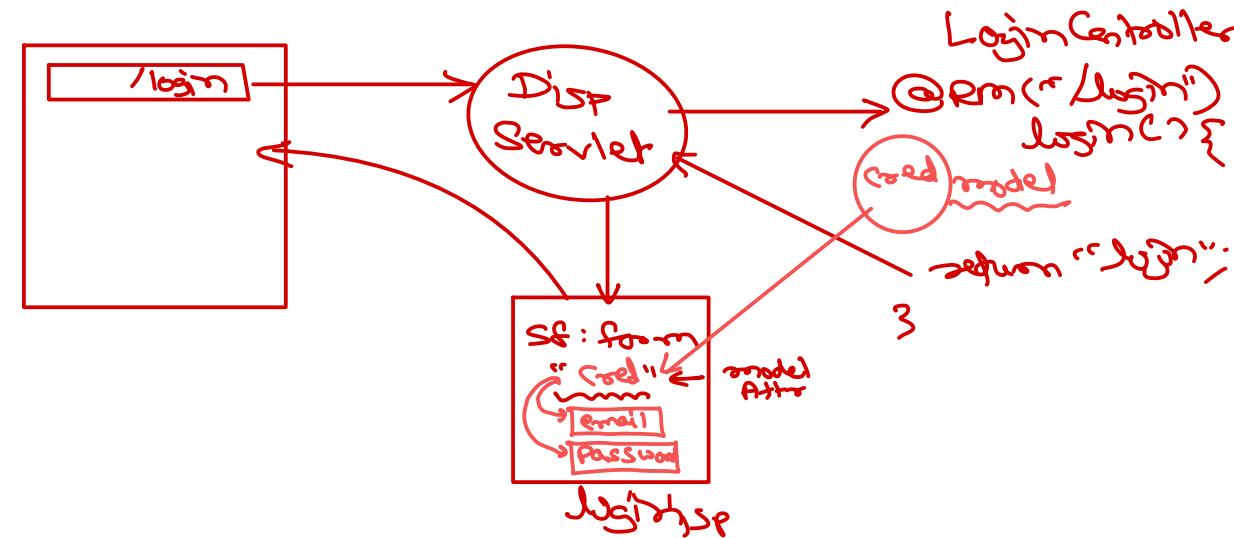
# Request handler method

- An @RequestMapping handler method can have a very flexible signatures.
- Supported method argument types
  - HttpServletRequest, HttpServletResponse
  - @RequestParam, @RequestHeader, @PathVariable
  - Map or Model or ModelMap → to send data to view.
  - Command object, @ModelAttribute, Errors or BindingResult
  - InputStream, OutputStream
  - HttpSession, @CookieValue, Locale
  - HttpEntity<>, @RequestBody.
- Supported method return types
  - String, View, Model or Map, ModelAndView
  - HttpHeaders, void
  - HttpEntity<>, ResponseEntity<>, @ResponseBody.



# Using spring tags

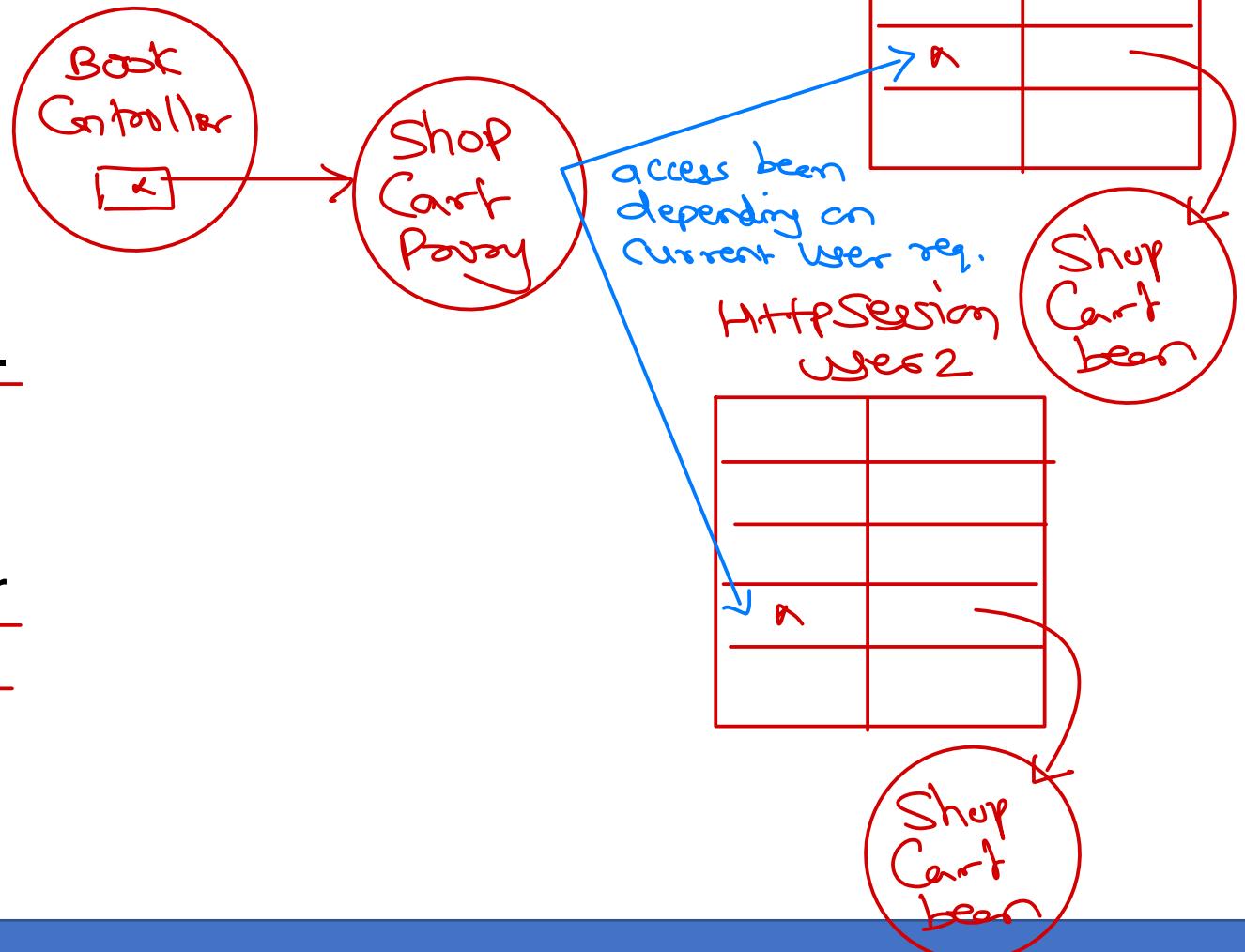
- Spring can work well with JSP and JSTL tags.
- Spring also have its own set of custom tags mainly for HTML input form, localization and validation.
- `<%@ taglib prefix="sf"  
uri="http://www.springframework.org/tags/form" %>`
- `<sf:form modelAttribute="command" action="auth">` ✓
  - Form backing bean or command object.
    - `<sf:input path="email"/>`
    - `<sf:password path="password"/>` ✓
- The *command* is model (POJO) object that carry data between view and controller.
- Model objects are request scoped.



# Session and Request scoped beans

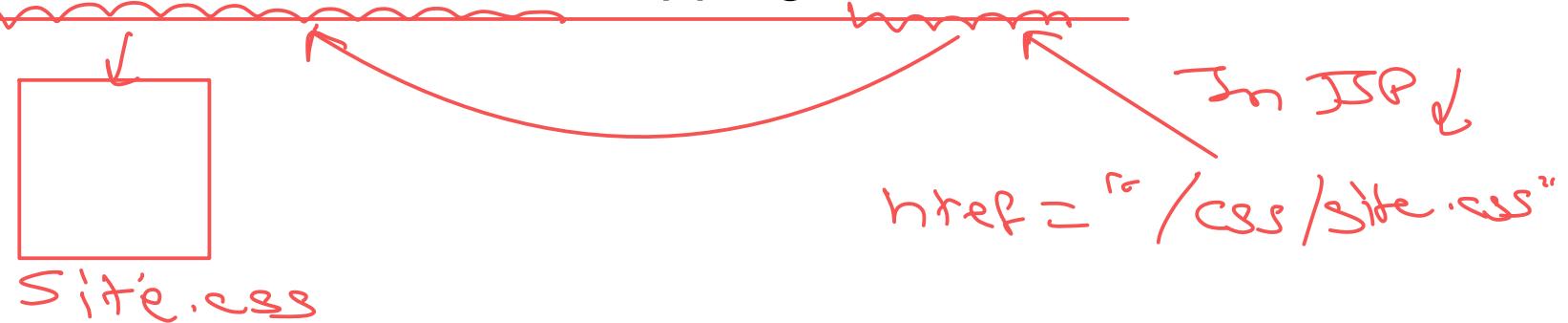
HttpSession  
User1

- Spring bean scopes
  - Singleton
  - Prototype
  - Session
  - Request
- Session and Request scope beans are only possible in web applications.
- The scope management is done via proxies. (internally)
- Bean proxy is auto-wired in controller class. Depending on session/request internally new bean is created and made available in that context.



# Spring Static resources

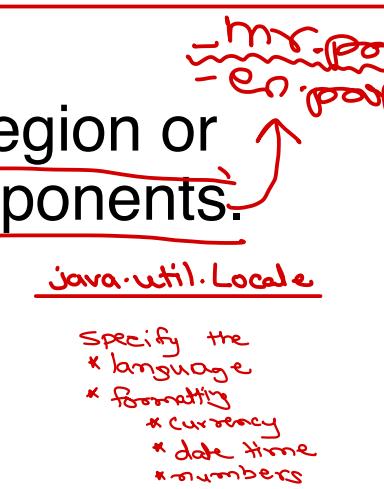
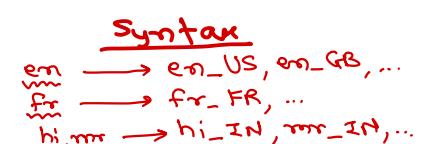
- Static resources like JS, CSS, images should be mapped to some location.
- <mvc:resources location="/WEB-INF/static/css/" mapping="/css/\*\*"/>



Spring Boot auto configures path of static resources  
to resources/static folder.

# Internationalization / Localization

- Adapting computer software to different languages, regional peculiarities and technical requirements of a target locale.
- Internationalization is the process of designing a software application so that it can be adapted to various languages and regions. → `mess.properties`  

- Localization is the process of adapting internationalized software for a region or language. Localization is done for each locale by added respective components.  

- Current locale can be accessed in request handler as Locale argument.
- Spring application steps  
  - Create messageSource bean and provide base path of properties file.
  - Create properties file for different locale.  


en	→ en_US, en_GB, ...
fr	→ fr_FR, ...
hi	→ hi_IN, mr_IN, ... :
  - In views add <s:message code="property-key-name"/>. 
- Current locale can be stored using SessionLocaleResolver or CookieLocaleResolver.
- Locale can be changed dynamically using LocaleChangeInterceptor (configured using <mvc:interceptors/>).



# Spring MVC validation

- For web applications client side validations are preferred for better user experience
- However client side validations can be bypassed/skipped by client.
- To ensure only validity of data provide server side validations (as well).
- Spring validation framework helps for server side validations.
- Spring supports JSR-303 validations.
  - @NotBlank, @NotEmpty, @Size, @Email, @Pattern, @DateTimeFormat, @Min, @Max
- Steps:
  - Add dependency hibernate-validator in pom.xml *(In spring boot → validator starters)*
  - Use appropriate annotations on model classes.
  - Use @Valid on @ModelAttribute in request handler method.
  - Next argument in request handler should be BindingResult method.
  - Use <sf:errors/> in view to show error codes.



# File upload and download

- File upload
  - Add commons-fileupload dependency in pom.xml.
  - In spring mvc config add CommonsMultipartResolver bean.
  - In view form, add HTML file input tag. Make form enctype="multipart/form-data", method="post" and mention action of spring request handler.
  - In request handler take arg @RequestParam("file") CommonsMultipartFile file.
  - File data will be accessible using file.getBytes(). Process that data.
- Hibernate blob handling
  - Create table with binary data as BLOB type.
  - In entity class map column with @Column and @Lob to byte[].
- File download
  - In spring request handler take HttpServletResponse as arg and return type void.
  - resp.setContentType(" application/octet-stream");
  - resp.getOutputStream().write(byteArray);





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

# Java EE

## Agenda

- Spring MVC advanced features
  - State management
  - CSS/Javascript/Images
  - Validation
  - Localization
  - File upload
- JSP limitations
- Thymeleaf Introduction

## Spring Security

### Password Encryptions

- Ensure that password column in database can store encrypted Password.

```
alter table customers modify column password varchar(200);
```

- In pom.xml add Spring Security Starter.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

- Disable default spring security config in main class.

```
@EnableAutoConfiguration(exclude = SecurityAutoConfiguration.class)
```

- Create a config class for security com.sunbeam.config.SecurityConfig with @Configuration.
- In this class create passwordEncoder bean.
- While signup encrypt the password.
  - CustomerServiceImpl auto-wire passwordEncoder bean.
  - Before save(), encrypt are set the password.
- While signin compare encrypted password in authenticate method.
  - Use passwordEncoder.matches(rawPassword, encPassword).
  - rawPassword -- collected from user in login page.
  - encPassword -- taken from database.

# Spring MVC

## State Management

- Spring allows server side state management and client side state management.
  - HttpSession attributes
    - HttpSession session --> session.setAttribute() and session.getAttribute() can be used.
    - Spring bean with @Scope("session").
  - Request attributes --> spring Model is abstraction on request attributes.
    - To send data from controller to the view.
      - Model model --> model.addAttribute("key", value) and In JSP \${key}
      - Map<String, Object> model --> model.put("key", value) and In JSP \${key}
    - Also req.setAttribute() and req.getAttribute() can also be used.
    - Spring bean with @Scope("request").
  - Cookies --> resp.addCookie("key", "value") and @Cookie("key") String value
  - QueryString --> @RequestParam or Command object
  - Hidden Form Fields --> <input type="hidden" ...> and @RequestParam or Command object

## Spring beans scopes

- Spring bean has four possible scopes
  - singleton, prototype, request and session.

## Implementing session beans

- Write a spring bean class to store the desired state and declare it's scope as session.

```
interface ShoppingCart {
    // ...
}

@Scope("session")
@Component
public class ShoppingCartImpl implements ShoppingCart {
    private List<Integer> cart;
    // ctor, getter/setter
}
```

- Autowire shoppingCart bean in the required controller.

```
public class BookControllerImpl {
    @Autowired
    private ShoppingCart shoppingCart;
    // ...
    @RequestMapping("/showCart")
    public String showCart() {
        for(int id: shoppingCart.getCart()) {
```

```
// ...
}
// ...
}
}
```

- Internally ShoppingCart shoppingCart; is created as a singleton proxy object. Depending on current user corresponding HttpSession object is accessed to save the cart.

## Spring Form Tags

- Spring can work with HTML tags.
- For advanced features like validation, localization you should use spring form tags.
- Refer slides for general steps to use Spring form tags.
  - 1. Add <%@ taglib ... %>
  - 2. <sf:form ModelAttribute="..." ...>
  - 3. <sf:input path="fieldName" ...>

## Spring MVC Validations

- End user may enter wrong data while registration/sign in/data entry.
- In "general" there are two ways of Validations
  - Client side Validations
    - <input type="text" required>
    - JS or jQuery framework for validation
    - Client side validations are faster and gives better user experience.
    - Client side validations can be compromised by the end user (disable JS, ...).
  - Server side Validations
    - Spring MVC supports server side validations using spring form tags.
    - It uses standard annotations for the data validations (JSR 303).
      - @NotBlank -- should not be empty (primitive types and string)
      - @NotNull -- should not be null
      - @Min and @Max -- numeric values like age.
      - @DateTimeFormat -- desired date format
      - @Pattern -- Regular expression
      - @Size -- string length
      - @Email -- Email pattern check (internally use regex)
    - To process validations use @Valid on command/pojo object.
    - In Spring MVC also use @ModelAttribute and refer the object into spring form.
    - To collect the validation errors use BindingResult or Errors object as next argument (after command object).

## Spring MVC Static Resources

- Create folder structure for JS, CSS and/or Images under resources/static folder.
- Create required js, css or image files into respective folders.
- Use their links into JSP pages.
  - Example1:

- Example2:
- Example3: 

## File Upload/Download

### Demo 20

- New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Web
  - Finish
- Create folder structure src/main/webapp/WEB-INF/jsp for JSP pages.
- However, in this application we are keeping files into webapp. (Directly accessible to user).
- In application.properties file set JSP prefix and suffix.

```
spring.mvc.view.prefix=/  
spring.mvc.view.suffix=.jsp
```

- You can also define limitations for the file upload (Optional - default settings will be followed otherwise.)

```
spring.servlet.multipart.max-file-size=4096KB  
spring.servlet.multipart.max-request-size=4096KB
```

- Create JSP page src/main/webapp/index.jsp for uploading file.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"  
    pageEncoding="ISO-8859-1"%>  
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="ISO-8859-1">  
<title>Index</title>  
</head>  
<body>  
<form method="post" enctype="multipart/form-data" action="upload">  
<input type="file" name="uploadFile"/>  
<br/>  
<input type="submit" value="Upload File"/>  
</form>  
</body>  
</html>
```

- Create controller com.sunbeam.controllers.FileControllerImpl for upload and download actions.

- Write request handler method for file uploading.

```
SECRET
@RequestMapping("/upload") // corresponding to index.jsp action=upload
public String upload(@RequestParam("uploadFile") MultipartFile file, // In
Spring Boot use MultipartFile instead of CommonsMultipartFile. "uploadFile"
is name of file control in prev page.
    Model model // to send info to the next view
) {
    // get uploaded file's details and add into model attributes (to show in
info.jsp)
    String name = file.getOriginalFilename();
    model.addAttribute("name", name);
    model.addAttribute("size", file.getSize());
    model.addAttribute("type", file.getContentType());
    //USE file.getBytes() or file.getInputStream() to access file contents
    // I've hard-coded server upload dir to "D:/test/". Please change as per
your system path.
    try(FileOutputStream out = new FileOutputStream("D:/test/"+name)) {
        // copy from uploadFile to the file on server disk
        FileCopyUtils.copy(file.getInputStream(), out);
        model.addAttribute("status", "Success");
    }catch (Exception e) {
        e.printStackTrace();
        model.addAttribute("status", "Failed");
    }
    return "info"; // view name: /info.jsp
}
```

- Create JSP page src/main/webapp/info.jsp to display the file information using EL syntax.
- Write request handler method for file download.

~~SECRET~~

```
@RequestMapping(value="/download", // same as img src=download into info.jsp
produces = "image/jpg") // assuming download file is jpg. Change
as per your file type
// Can also change manually in resp.setContentType() in the
method.
public void download(@RequestParam("name") String name, // name is taken as
req param img src=download?name=...
    HttpServletResponse resp) // write file on resp stream directly
{
    // I've hard-coded server upload dir to "D:/test/". Please change as per
your system path.
    try(FileInputStream in = new FileInputStream("D:/test/"+name)) {
        ServletOutputStream out = resp.getOutputStream(); // get servlet
output stream
        FileCopyUtils.copy(in, out); // copy from server disk file into
response body
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
// return type void, because response data is written directly into  
response stream  
}
```

- Into info.jsp add image tag to display image file

```

```

## Bookshop4 Demo

### Encrypted password

- Ensure that password column in database can store encrypted Password.

```
alter table customers modify column password varchar(200);
```

- In pom.xml add Spring Security Starter.

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

- Disable default spring security config in main class.

```
@EnableAutoConfiguration(exclude = SecurityAutoConfiguration.class)
```

- Create a config class for security com.sunbeam.config.SecurityConfig with @Configuration.
- In this class create passwordEncoder bean.
- While signup encrypt the password.
  - CustomerServiceImpl auto-wire passwordEncoder bean.
  - Before save(), encrypt are set the password.
- While signin compare encrypted password in authenticate method.
  - Use passwordEncoder.matches(rawPassword, encPassword).
  - rawPassword -- collected from user in login page.
  - encPassword -- taken from database.

## Spring Form Tags

- In login.jsp, change html tags to spring form tags.

```

<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Login</title>

</head>
<body>
    <h2>Online Book Shop</h2>
    <sf:form modelAttribute="cred" method="post" action="authenticate">
        <table>
            <tr>
                <td>Email:</td>
                <td><sf:input path="email"/></td>
            </tr>
            <tr>
                <td>Password:</td>
                <td><sf:password path="password"/></td>
            </tr>
            <tr>
                <td colspan="2">
                    <input type="submit" value="Sign In"/>
                    <a href="register">Sign Up</a>
                </td>
            </tr>
        </table>
    </sf:form>
</body>
</html>

```

- In LoginControllerImpl.login() method add empty Credentials object into model as "cred" modelAttribute.
- Run application. Check if login.jsp is displayed. Also see if Sign In functionality is working well.

## Spring Validations

- In pom.xml add validation starter.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>

```

- In model/pojo Credentials class use appropriate validation annotations.
- In request handler method -- LoginControllerImpl.authenticate()
  - The command/model object should be marked as @Valid.
  - In Spring MVC project, also use @ModelAttribute("...").
  - Immediate next argument must be BindingResult to collect validation errors.

- Check BindingResult object to see errors -- hasErrors(). Send user to current view if error is found.
- In login.jsp add `sf:errors` tags to display the error messages for email and password.
- Run application and check if validations are working.

## Static resources

- Create folder structure for Static resources i.e. css, js and images.
- Create file site.css into css directory. Define error class in it.
- Link file into login.jsp and check if css class is applied to error message.

```
<link rel="stylesheet" href="/css/site.css"/>
```

## Spring Localization

- Create file messages.properties under resources directory.
- In application.properties add "spring.messages.basename=messages". This optional because default name is messages.
- In login.jsp file display all labels from this file using `<s:message>`.

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="s" uri="http://www.springframework.org/tags" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Login</title>
<link rel="stylesheet" href="/css/site.css"/>

</head>
<body>
    <h2><s:message code="app.title" text="####"/></h2>
    <sf:form modelAttribute="cred" method="post" action="authenticate">
        <table>
            <tr>
                <td><s:message code="email.label" text="####"/></td>
                <td><sf:input path="email"/></td>
                <td><sf:errors path="email" cssClass="error"/></td>
            </tr>
            <tr>
                <td><s:message code="password.label" text="####"/></td>
                <td><sf:password path="password"/></td>
                <td><sf:errors path="password" cssClass="error"/></td>
            </tr>
            <tr>
                <td colspan="2">
                    <input type="submit" value="<s:message
code="signin.label" text="####"/>"/>
                    <a href="register"><s:message code="signup.label"
text="####"/></a>
                </td>
            </tr>
        </table>
    </sf:form>
</body>

```

```
</td>
</tr>
</table>
<%-
<sf:errors path="*"/>
--%>
</sf:form>
</body>
</html>
```

- Run application and check if default messages are visible (from messages.properties).
- Create files messages\_hi.properties, messages\_mr.properties and messages\_en.properties as discussed in class.
  - Use translator to get corresponding messages.
  - Copy them directly into eclipse. Eclipse will convert it into UTF-8 format.
- In login.jsp add

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

## Bookshop Cart functionality

- These are basic MVC steps with same logic as of servlets.
- Please follow them from video (if you don't recall them).

## JSP limitations

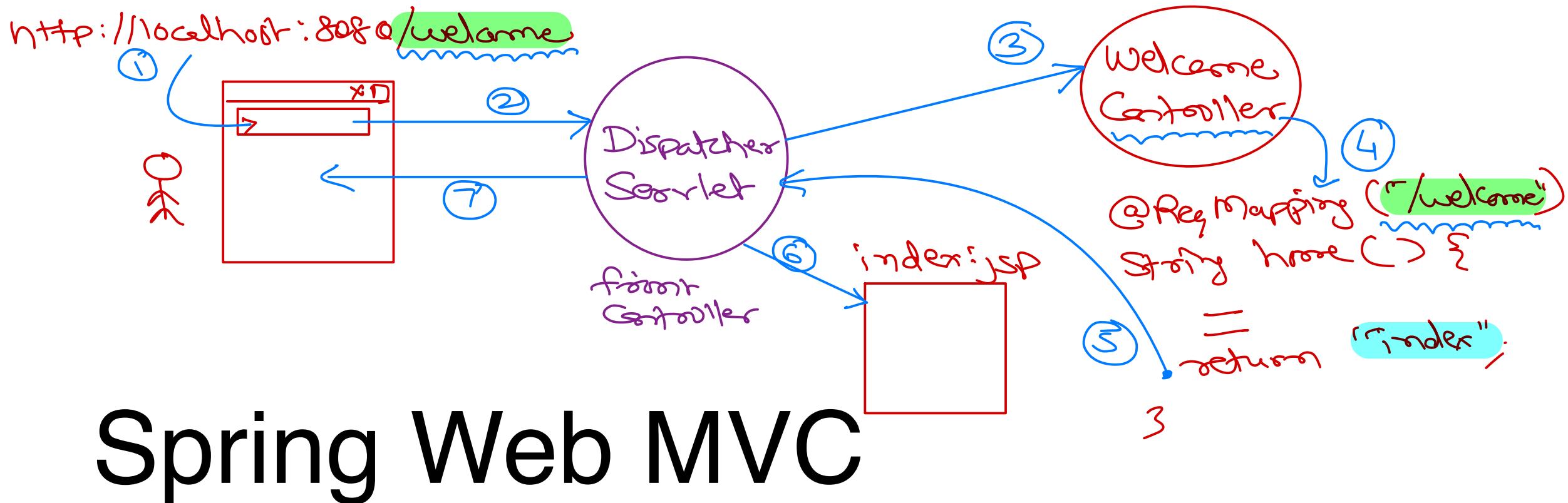
- Spring boot runs with "embedded" web server like tomcat/jetty. With executable jar, JSP files are not accessible directly.
- JBoss Undertow web server does not support JSPs.
- Creating a custom error.jsp page does not override the default view for error handling. Custom error pages should be used instead.



# Spring and Hibernate

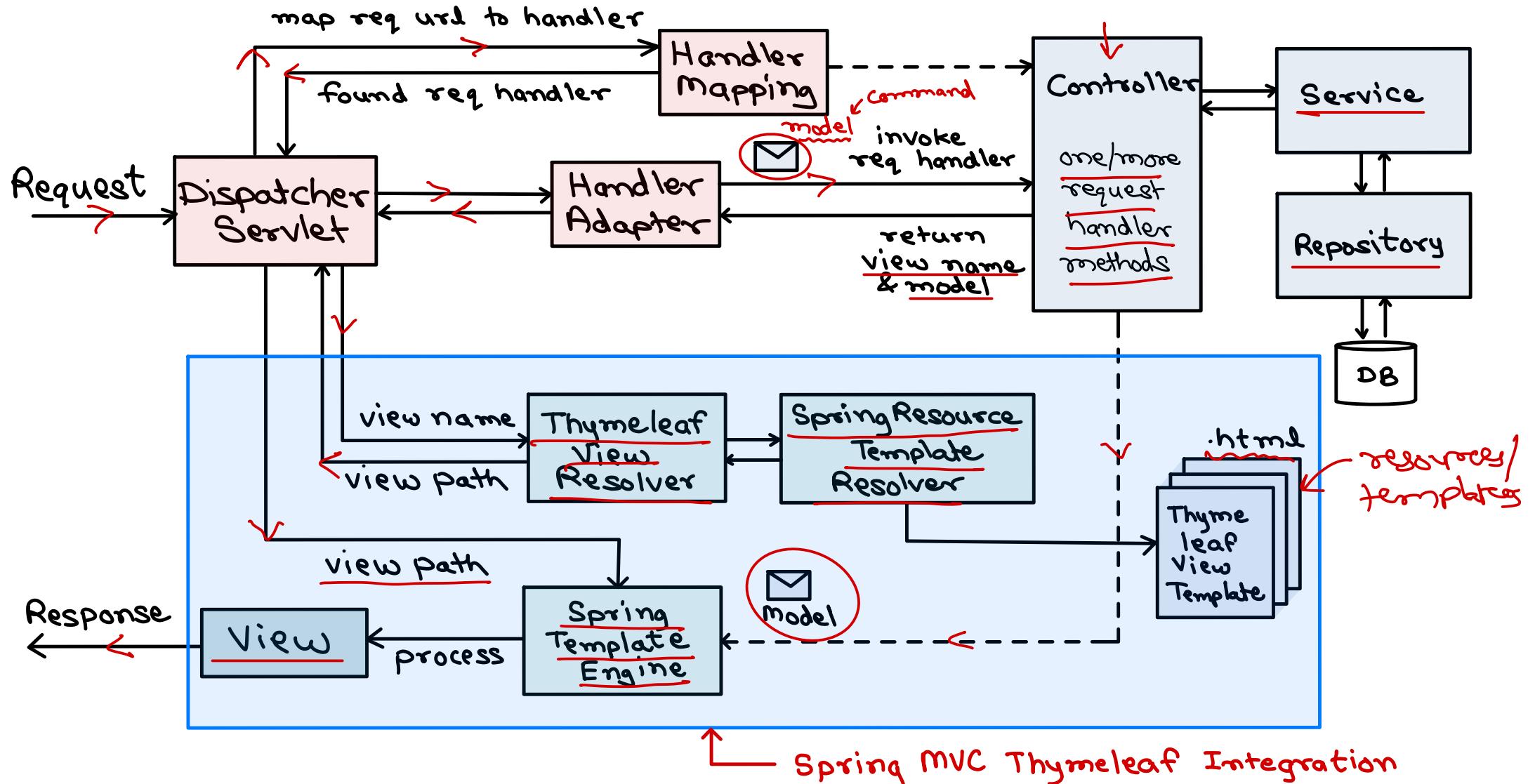
*Nilesh Ghule <nilesh@sunbeaminfo.com>*





# Spring Web MVC

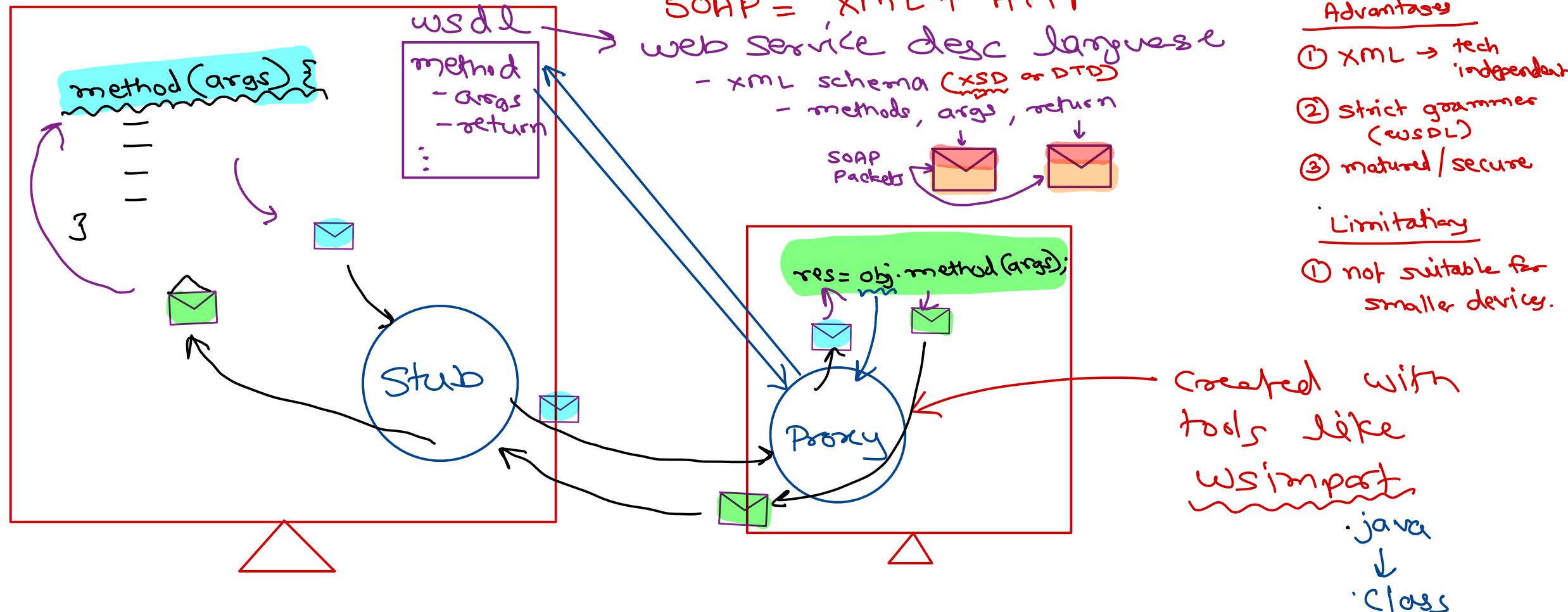
# Spring Web MVC + Thymeleaf



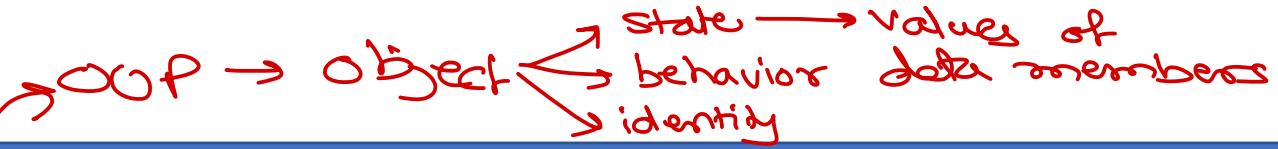
# REST services



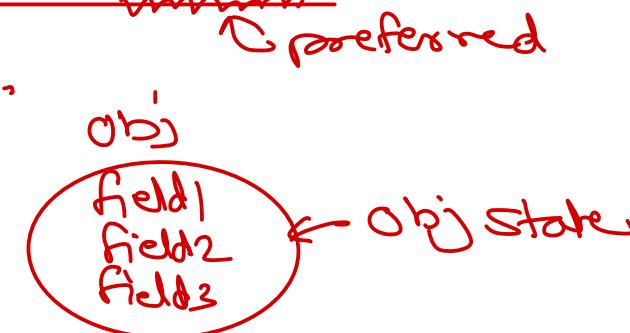
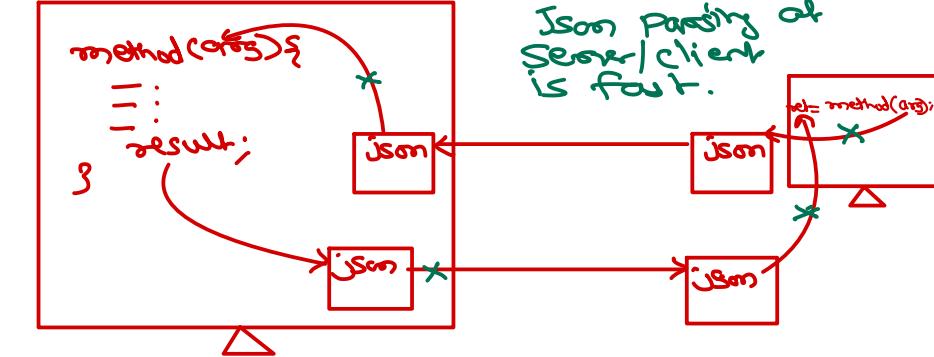
# XML based services - SOAP (Simple Object Access Protocol)



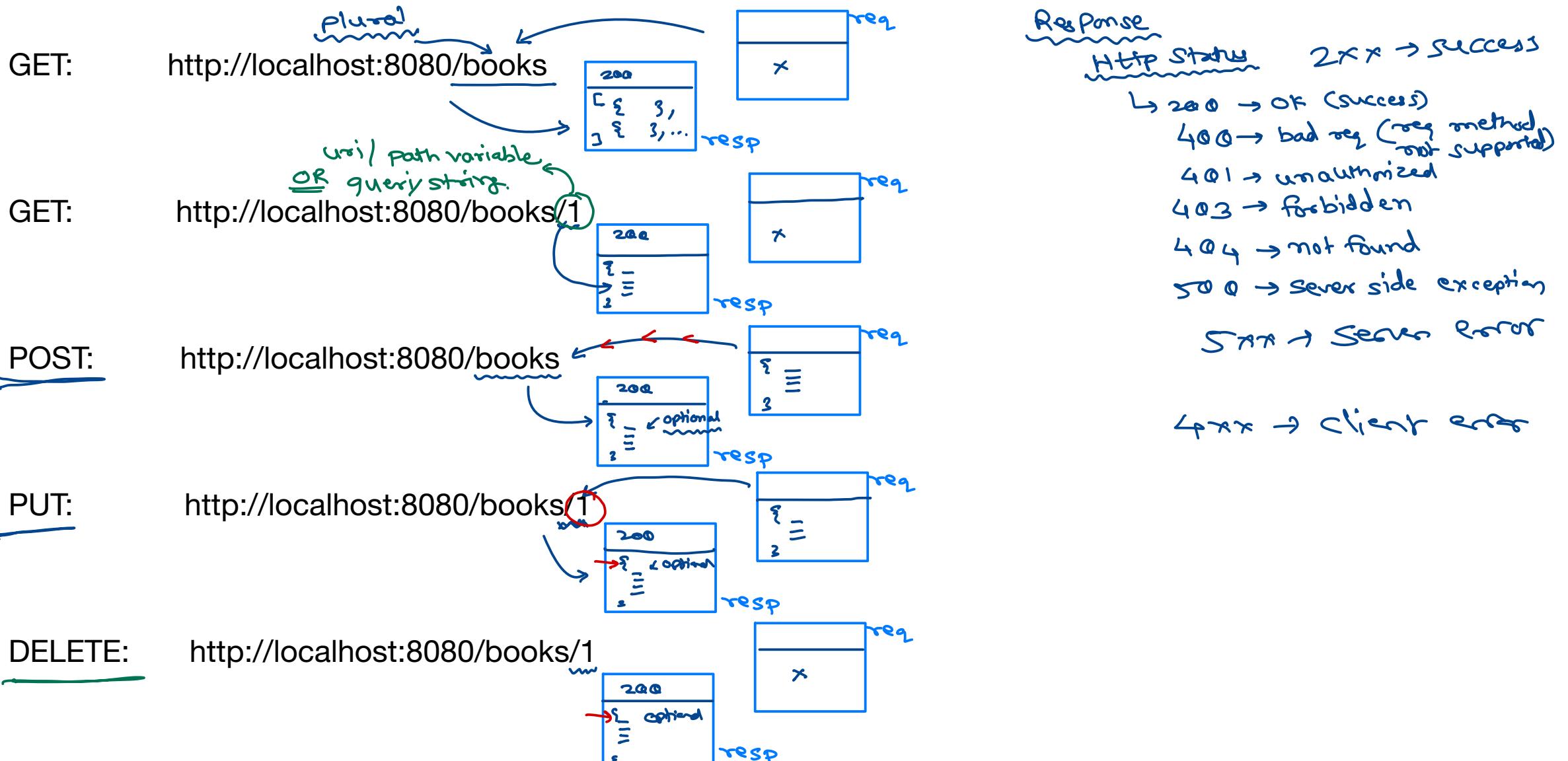
# REST services



- REpresentation State Transfer
- Protocol to invoke web services from any client.
  - Client can use any platform/language.
    - Java
    - .Net
    - PHP
    - C/C++
- REST works on top of HTTP protocol.
  - Can be accessed from any device which has internet connection.
  - REST is lightweight (than SOAP) – XML or JSON.
  - Uses HTTP protocol request methods
    - GET: to get records
    - POST: to create new record
    - PUT: to update existing record
    - DELETE: to delete record



# REST Services Convention

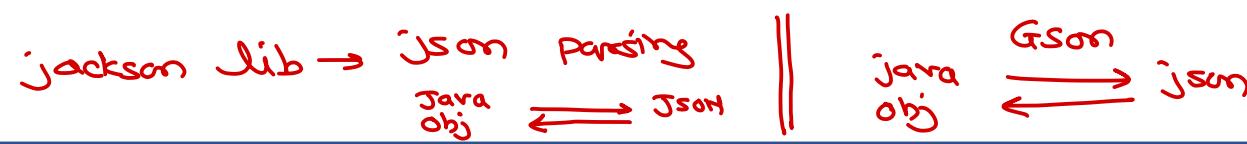


# Spring REST

---



# Spring REST services

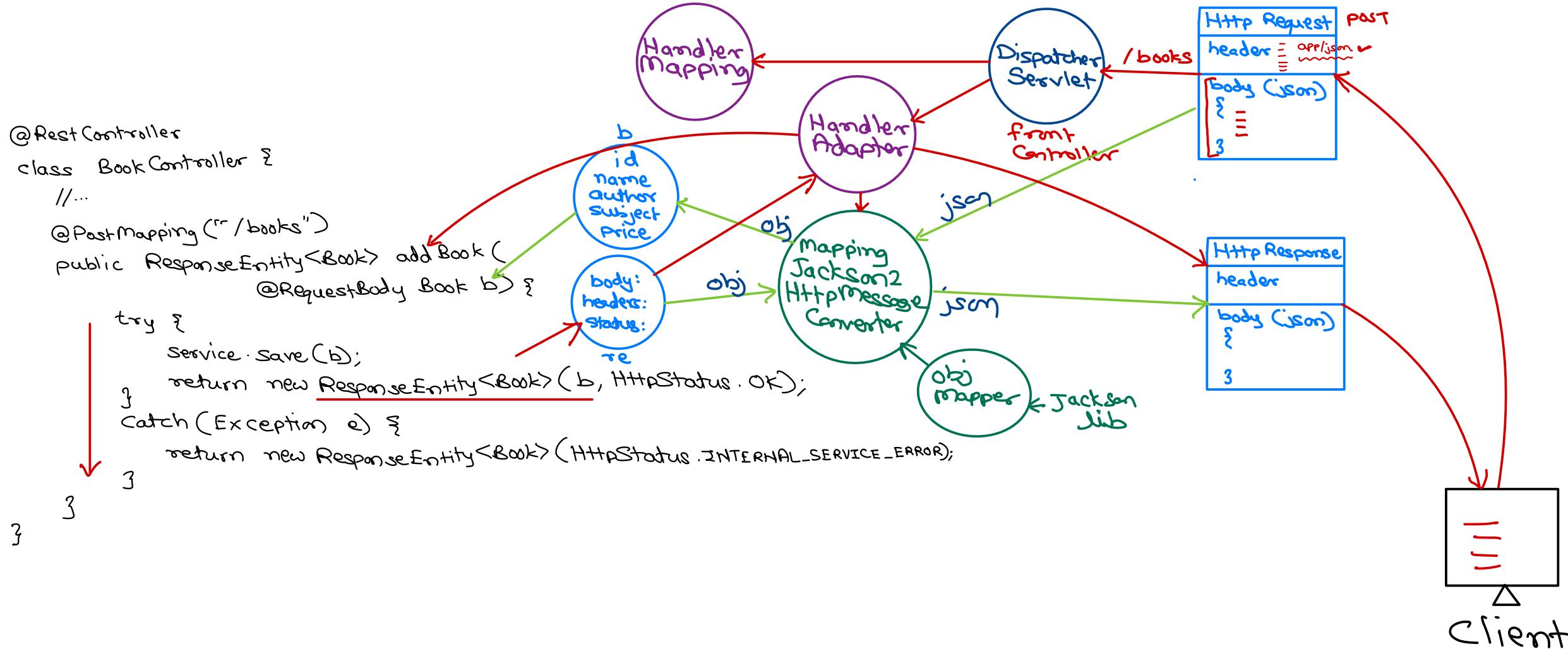


- Based on top of Spring Web MVC.  
↳ Strong conv → for Spring boot  
↳ opinionated default  
↳ web-starter
- Maven dependency: jackson-databind and jackson-databind-xml (if need XML)
- HttpMessageConverter beans:
  - MappingJackson2HttpMessageConverter, MappingJackson2XmlHttpMessageConverter
  - @Override configureMessageConverters() → MVC Config → auto create msg converter beans.
- Using @Controller
  - @GetMapping, @PostMapping, @PutMapping, @DeleteMapping or @RequestMapping
  - @RequestBody, @ResponseBody / ResponseEntity<>
  - @PathVariable, @RequestParam
- Using @RestController
  - @GetMapping, @PostMapping, @PutMapping, @DeleteMapping or @RequestMapping
  - @RequestBody, ResponseEntity<>
  - @PathVariable, @RequestParam
- To manipulate JSON response use @JsonProperty or @JsonIgnore.



# Spring REST services

```
@RestController  
class BookController {  
    ...  
    @PostMapping("/books")  
    public ResponseEntity<Book> addBook(  
        @RequestBody Book b) {  
        try {  
            service.save(b);  
            return new ResponseEntity<Book>(b, HttpStatus.OK);  
        } catch (Exception e) {  
            return new ResponseEntity<Book>(HttpStatus.INTERNAL_SERVER_ERROR);  
        }  
    }  
}
```



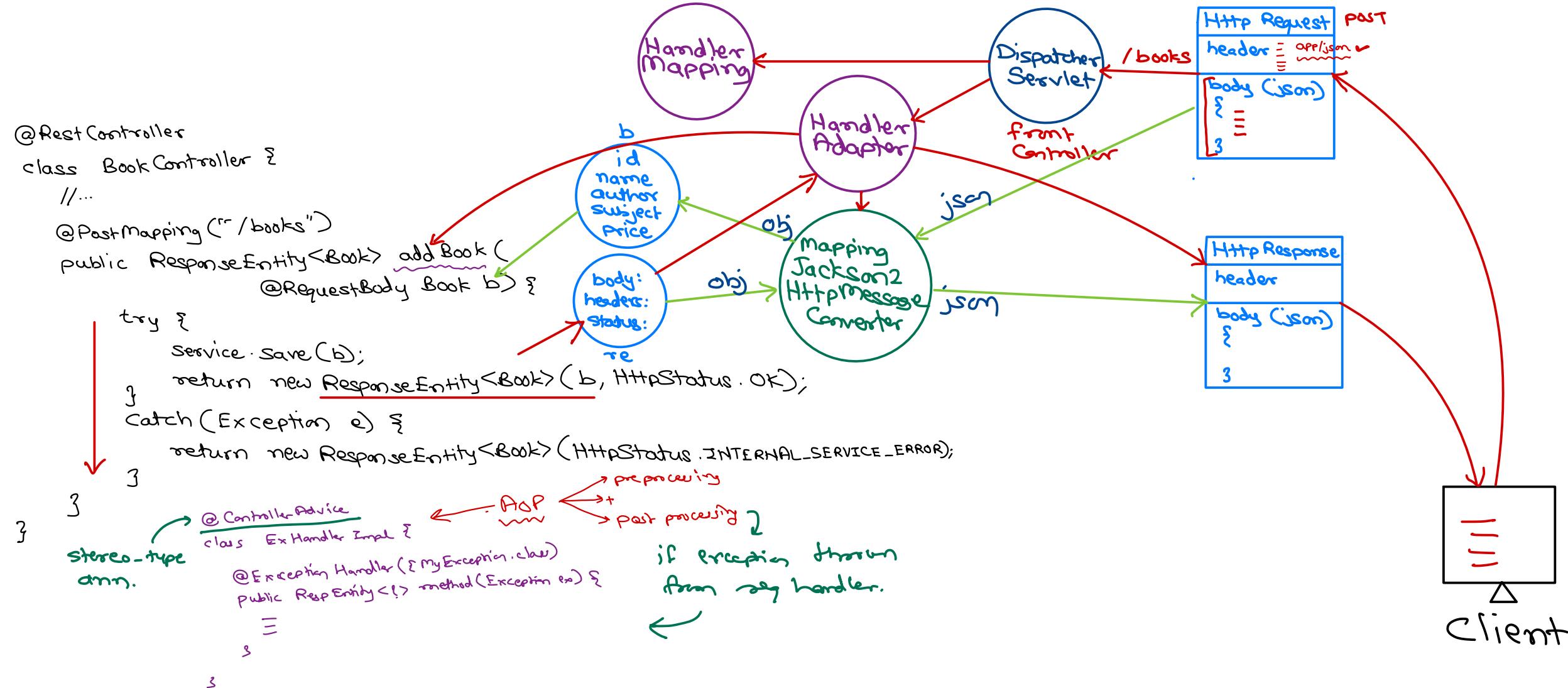
# Spring REST services

```
@RestController  
class BookController {  
    ...  
    @PostMapping("/books")  
    public ResponseEntity<Book> addBook(@RequestBody Book b) {  
        try {  
            service.save(b);  
            return new ResponseEntity<Book>(b, HttpStatus.OK);  
        } catch (Exception e) {  
            return new ResponseEntity<Book>(HttpStatus.INTERNAL_SERVER_ERROR);  
        }  
    }  
}
```

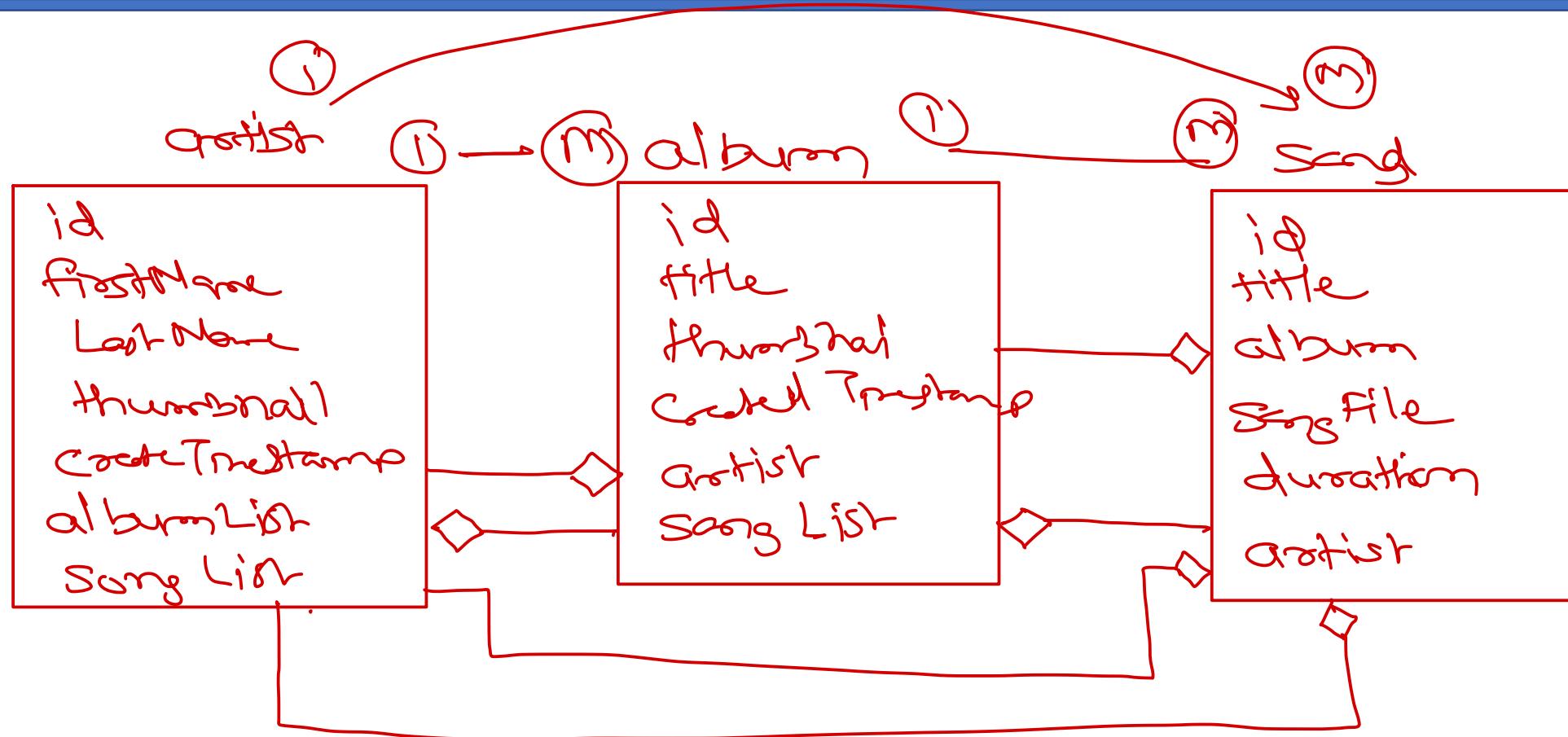
stereotype  
ann.

```
@ControllerAdvice  
class ExHandlerImpl {  
    @ExceptionHandler({myException.class})  
    public ResponseEntity<!> method(Exception ex) {  
        ...  
    }  
}
```

3  
3  
3



# Spring React Example - gaana.com





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

# Java EE

## Agenda

- Thymeleaf Introduction
- SOAP protocol
- REST protocol
- Spring REST services
- Spring REST client
- Spring REST React Integration

## Thymeleaf - QuickStart

- Not part of Spring IO platform
- Templating engine
- Request and response
  - Client ----> Controller ----> View ----> Template Engine - Render ----> Response ----> Client
- `<html xmlns:th="http://www.thymeleaf.org">`
- `<p th:text="'Hello World'">`
  - 'single quotes' for string constants
- Eclipse Marketplace - Thymeleaf plugin (optional)

## Thymeleaf vs JSP

- Thymeleaf & JSP both can be used to build views in web applications.
- Thymeleaf can be used in non-web context as well.
  - Email template
  - PDF template

## Spring MVC Thymeleaf - Hello World!

- step 1: Create new spring starter project with following dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

- step 2: Add HomeController (under src/main/java - current package)

```
@Controller  
public class HomeController {  
    @RequestMapping("/index") // url --> handlerMapping  
    public String index(Model model) {  
        model.addAttribute("curTime", new Date().toString());  
        return "home"; // view name --> viewResolver  
    }  
}
```

- step 3: Create thymeleaf template (under src/main/resources/templates - home.html)

```
<!DOCTYPE html>  
<html xmlns:th="http://www.thymeleaf.org">  
<head>  
    <meta charset="ISO-8859-1">  
    <title>Home</title>  
</head>  
<body>  
    <h3>Welcome to Home</h3>  
    <span th:text="${curTime}"></span>  
</body>  
</html>
```

- step 4: Run As - Spring Boot application
- step 5: In browser - http://localhost:8080/index

## Thymeleaf config

- ViewResolver configuration is "by default" in Spring Boot. Can be overridden as follows:

```
@Configuration  
@EnableWebMvc  
public class ThymeleafConfiguration {  
    @Bean  
    public ThymeleafViewResolver thymeleafViewResolver() {  
        ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();  
        viewResolver.setTemplateEngine(templateEngine());  
        resolver.setContentType("text/html");  
        return viewResolver;  
    }  
  
    @Bean  
    public SpringTemplateEngine templateEngine() {  
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();  
        templateEngine.setTemplateResolver(thymeleafTemplateResolver());  
        return templateEngine;  
    }
```

```
@Bean  
public SpringResourceTemplateResolver thymeleafTemplateResolver() {  
    SpringResourceTemplateResolver templateResolver = new  
    SpringResourceTemplateResolver();  
    templateResolver.setPrefix("/WEB-INF/views/");  
    templateResolver.setSuffix(".html");  
    templateResolver.setTemplateMode(TemplateMode.HTML);  
    return templateResolver;  
}  
}
```

## REST

- Refer slides.

### REST service

- New -- Spring Starter Project
  - Fill project details
  - Spring Starter Dependencies
    - Spring Data JPA
    - MySQL Driver
    - Spring Web
    - Spring Dev Tools
  - Finish
- In application.properties, add database/hibernate settings.
- Create entities, daos and services.
- Create controller classes.
  - If using @Controller, then **request** handler methods returning pojo classes should be marked with @ResponseBody.
  - If using @RestController, @ResponseBody is optional while returning Pojo.
  - Ususally controller base path is written on top of controller class using @RequestMapping e.g. @RequestMapping("/books").
  - Typically ResponseEntity<> is returned from request handler method.
    - ResponseEntity is wrapper around pojo object to return.
    - It also helps managing status and response headers.
    - Have helper methods to build response like ok(), status(), etc.
    - <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/ResponseEntity.html>
  - The request methods are marked with @GetMapping, @PostMapping, @PutMapping and @DeleteMapping.
    - Refer slides (REST conventions) for standard arguments are return values.
  - Use @RequestBody if input data is sent (from client) in request body (ususally json). If using form-data, do not use @RequestBody.
  - Note that Jackson converter work for request body and response body (but not for request param / form data).
  - To return customized json there are two ways.

- Create pojo class as per desired JSON.
- Create Map<String, Object> with desired keys.
- DTO classes
  - Rest controllers can return entity objects directly.
  - However standard practices suggest not mixing data layer with frontend layer.
  - DTO classes are used to send/retrieve data from frontend.
  - They should be designed as per client application requirement.
  - Programmer can use @JsonProperty, @JsonIgnore, etc jackson annotations to control produced json.
  - Conversion from entity to DTO can be done in one of the following ways
    - Use DTO constructor.
    - Use static helper methods like fromEntity() and toEntity() or implement custom converter classes for the same.
    - Helper methods can do conversions manually or use spring BeanUtils.copyProperties(). Note that copyProperties() can copy only matching properties; remaining properties should copied manually.
    - Third party libraries can also be used for entity to DTO conversions e.g. ModelMapper, MapStruct, etc. <https://www.baeldung.com/java-performance-mapping-frameworks>



Java EE  
Sunbeam Infotech





# Java Server Pages

Sunbeam Infotech



# JSP – Custom tags

- ① JSP actions
- ② JSTL tags
- ③ third-party tags.

replace java code in jsp scriptlet.

- Custom tags are used to combine presentation logic and business logic.
- There are two types of tags
  - ✓ Classic tags: inherited from Tag interface or TagSupport class.
  - ✓ Simple tags : inherited from SimpleTag interface or SimpleTagSupport class.
- SimpleTag implementation steps
  - Decide tag name, attributes & body type.
  - Implement tag handler class inherited from SimpleTagSupport
    - ✓ Constructor (param less)
    - ✓ Fields & getter/setter = attributes
    - ✓ setJspBody() = if there is body
    - ✓ doTag() = Logic implementation
  - Implement .tld file to define tag syntax.
  - In JSP, use `<%@ taglib ... %>` & tag.

## SimpleTag life cycle

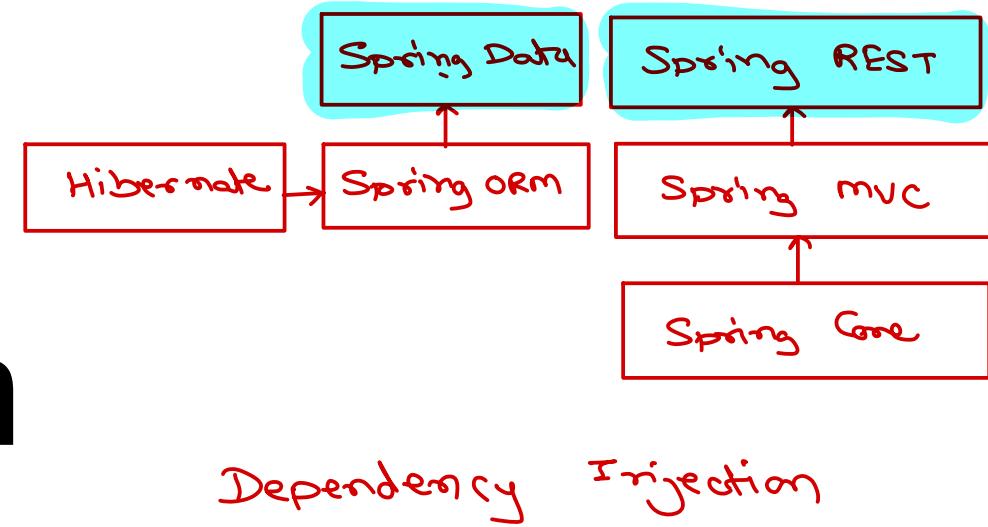
- A new tag handler instance is created each time by the container using default constructor.
- The `setJspContext()` and `setParent()`\* methods are called.
- The setters for each attribute defined for tag.
- If a body exists, the `setJspBody()` method is called.
- The `doTag()` method is called.
- The `doTag()` method returns and all variables are synchronized.

JSP  
empty

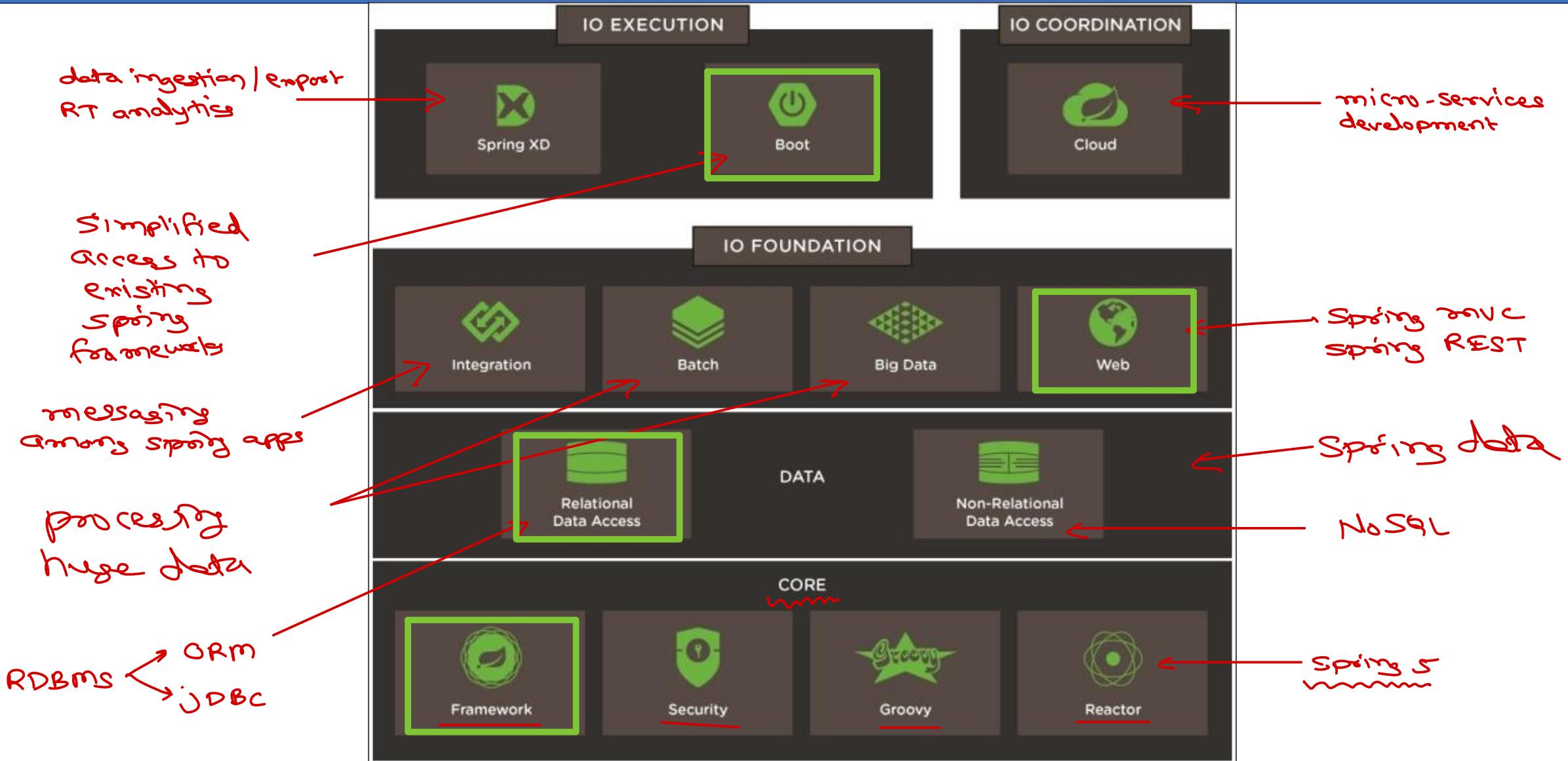
url : String  
delay : int  
(.tld)  
tag library  
descriptor



# Spring Introduction



# Spring IO platform



# Spring Framework

spring-core (2mb)

Java syntax is simple.

Java developer is not

covering all aspects

huge apps  
testing  
deployment  
maintenance

- Spring is light-weight comprehensive framework to simplify Java development.
- Developed by Rod Johnson. Spring 3 added annotations while Spring 5 added reactive.  
2003
- Spring pros

- Inversion of control (Dependency injection)
- Test driven development
- Extensive but Flexible and modular
- Smooth integration with existing technologies
- Eliminate boilerplate code
- Extendable
- Maintainable

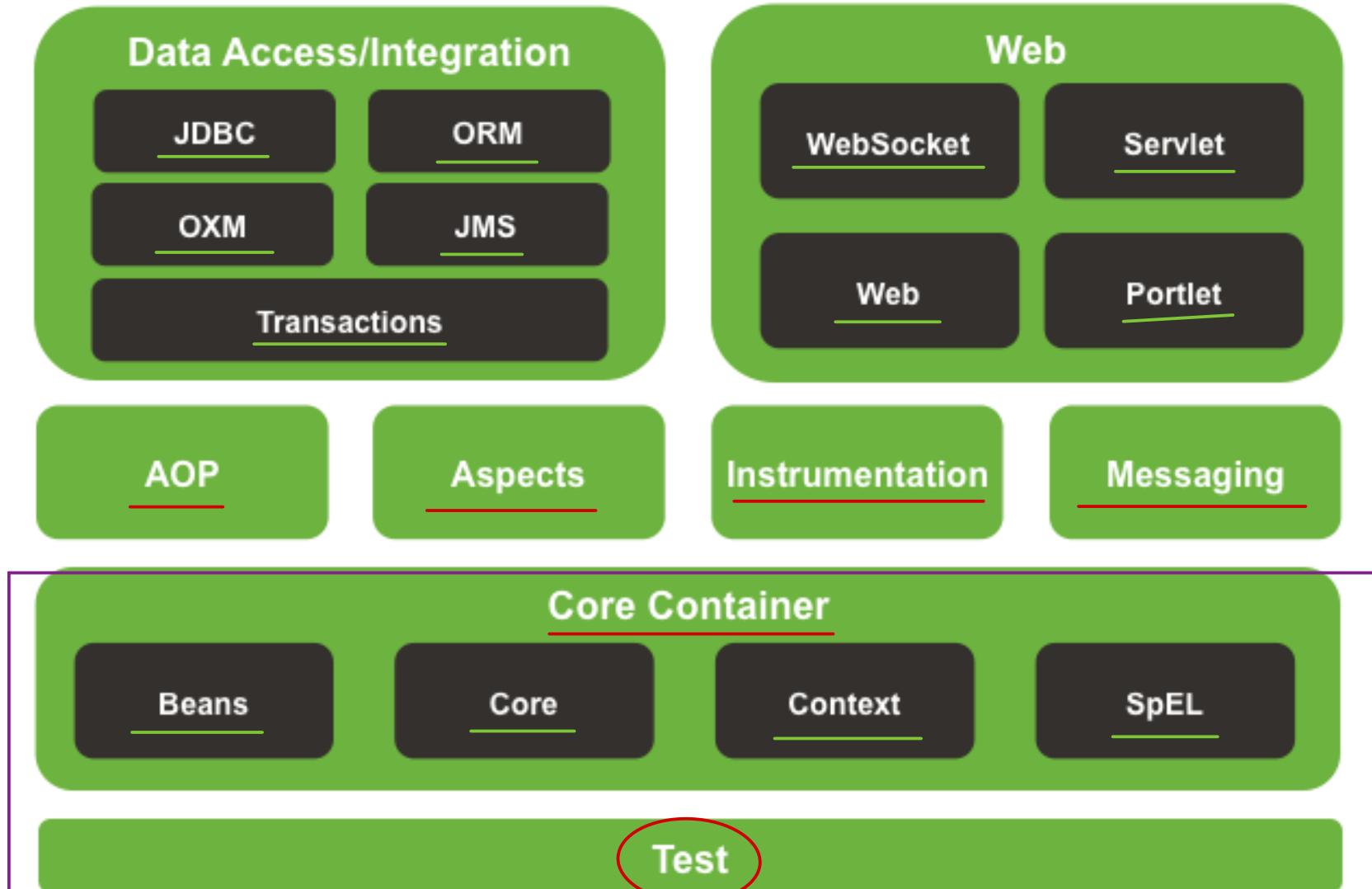
Struts : all - or - nothing  
Spring : flexible

## • Spring cons

- Heavy configuration (XML, Java, Both)
- Module versioning & compatibility
- Application deployment ↴



# Spring architecture (3.x)



# Spring Boot

- Spring Boot is NOT a new standalone framework. It is combination of various frameworks on spring platform i.e. spring-core, spring-webmvc, spring-data, ...
- Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”.
- Spring Boot take an "opinionated view" of the Spring platform and third-party libraries. So most of applications need minimum configuration.
- Primary Goals/Features
  - Provide a radically faster and widely accessible "Quick Start".
  - Opinionated config, yet quickly modifiable for different requirements.
  - Managing versions of dependencies with starter projects.
  - Provide lot of non-functional common features (e.g. security, servers, health checks, ...).
  - No extra code generation (provide lot of boilerplate code) and XML config.
  - Easy deployment and containerisation with embedded Web Server.
- Recommended to use for new apps and not to port legacy Spring apps.



# Spring Boot

## Spring Boot

Spring  
REST

Spring  
Session

Spring  
Batch

Spring  
Integration

...

Spring  
Data

Spring  
Big Data

Spring  
Security

Spring  
Social

Spring  
Kafka

Web

Database

Spring Framework

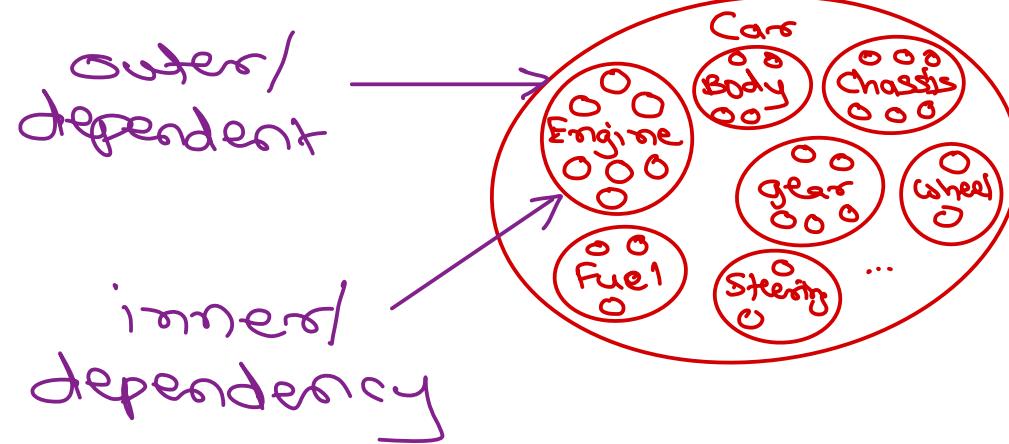
AOP

Messaging

**Spring Boot = Spring framework + Embedded web-server + Auto-configuration - XML config - Jar conflicts**



Composition  
has-a



Object 'instantiation' & 'initialization'

```
Car c = new Car();  
c.set Engine(e);  
c.set Wheels(ws);  
c.set Chassis(ch);  
c.set Gear(g);  
:  
c.drive();
```

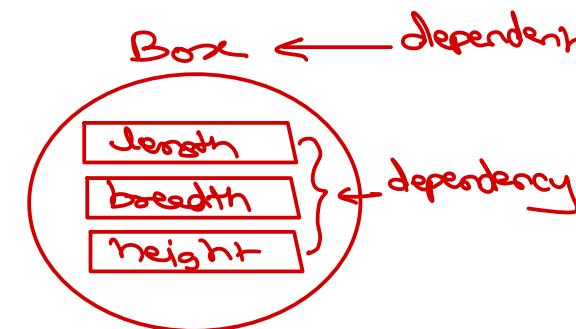
OOP : programmer  
manual      IOC  
Spring : container  
automated

Inversion of Control

# Spring Dependency Injection

# Dependency Injection

- Spring container injects dependency beans into dependent beans.
- Spring container is also called as IoC container.
- Dependency Injection
  - Setter based DI <property ...>
  - Constructor based DI <constructor-args ...>
  - Field based DI



```
Box b=new Box();
b.setLength(5);
b.setBreadth(4);
b.setHeight(3);
System.out.println(b.calcVolume());
```

```
Box b=ctx.getBean("box");
System.out.println(b.calcVolume());
```

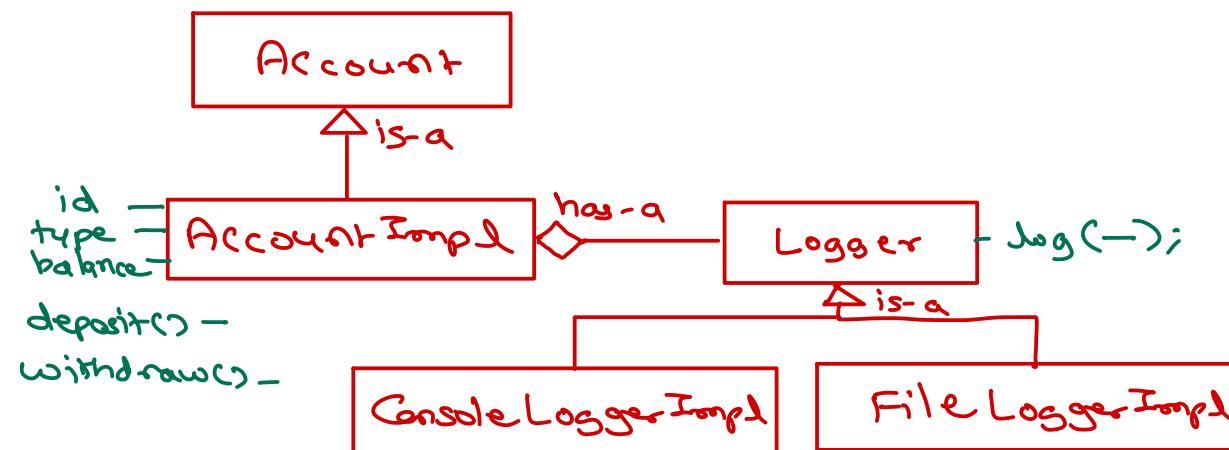
# XML based DI

- Spring beans are declared into *Spring bean configuration file.*
- Bean factory or application context reads the file. It instantiate and initialize bean objects at runtime.
- XML configuration
  - Initializing properties ✓
  - Initializing dependency beans ✓
  - Initializing collections ✓



# Dependency Injection

- One interface may have multiple implementations.
- This makes application more extendable and maintainable.
- Desired implementation can be plugged in using DI.



# XML Bean Factory

- Deprecated (*no more used*)
- Minimal Dependency injection features
  - Beans loading
  - Auto-wiring
- Create a singleton bean when it is accessed first time in the application.
- No bean life cycle management ✗



# ApplicationContext

- Features

- Dependency injection - Beans loading and Auto-wiring
- Automatic BeanPostProcessor registration i.e. Bean life cycle management
- Convenient MessageSource access (Internationalization)
- ApplicationEvent publication
- Create all singleton beans when application context is loaded.

impl  
classes →

- ApplicationContext

- ClassPathXmlApplicationContext
- FileSystemXmlApplicationContext
- AnnotationConfigApplicationContext
- WebApplicationContext
  - XmlWebApplicationContext
  - AnnotationConfigWebApplicationContext

base interface  
(refer spring doc)



# Annotation config

---

- Does spring configuration without using any XML file.
  - The bean creation is encapsulated in `@Configuration` class and beans are represented as `@Bean` methods.
  - Annotation configuration is processed using `AnnotationConfigApplicationContext`.
- 



# Spring beans

→ Beans created &  
managed by  
Spring Containers

Bean = Object of a java class

Java class = fields + constructors + getter/setters  
+ business logic



# Spring bean life cycle

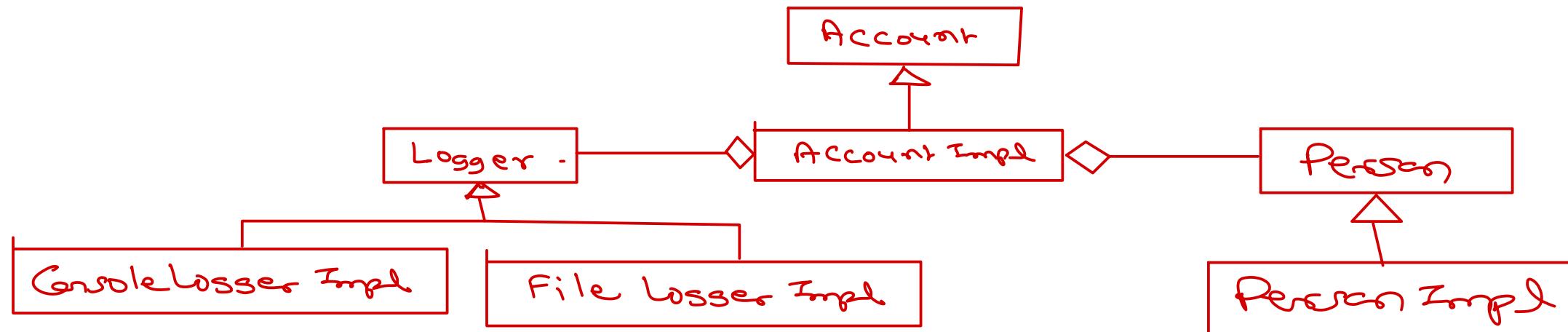
- Spring container creates (singleton) spring bean objects when container is started.
- Spring container controls creation and destruction of bean objects.
- There are four options to control bean life cycle.
  - InitializingBean and DisposableBean callback interfaces
  - Aware interfaces for specific behaviours
    - BeanNameAware, BeanFactoryAware, ApplicationContextAware , etc.
  - BeanPostProcessor callback interface
  - Custom init() / @PostConstruct and destroy() / @PreDestroy methods



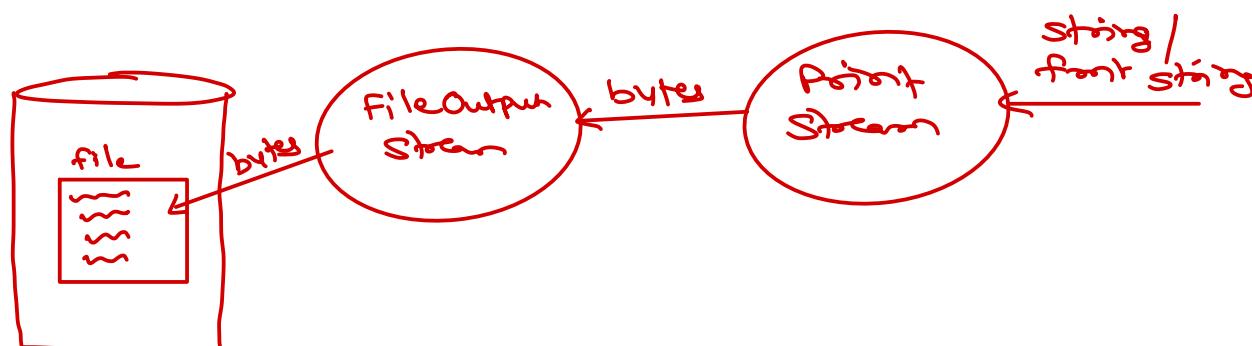
# Spring bean life cycle

- Bean creation
  - ✓ 1. Instantiation (Constructor)
  - ✓ 2. Set properties (Field/setter based DI)
  - ✓ 3. BeanNameAware.setBeanName()
  - ✓ 4. ApplicationContextAware.setApplicationContext()
  - ✓ 5. Custom BeanPostProcessor.postProcessBeforeInitialization()
  - ✓ 6. Custom init() (@PostConstructor)
  - ✓ 7. InitializingBean.afterPropertiesSet()
  - ✓ 8. Custom BeanPostProcessor.postProcessAfterInitialization()
- Bean destruction
  - ✓ 1. Custom destroy() (@PreDestroy)
  - ✓ 2. DisposableBean.destroy()
  - ✓ 3. Object.finalize()





# Stereo-type annotations



# Stereo type annotations

- Auto-detecting beans (avoid manual config of beans).

- @Component
- @Service
- @Repository
- @Controller and @RestController

@Controller for  
REST Services  
Spring 4.0 +

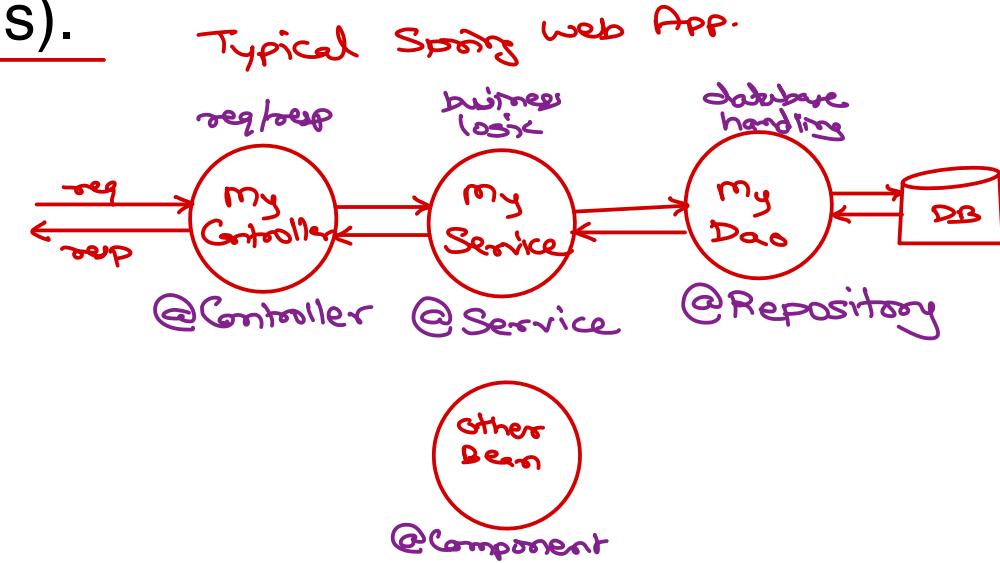
- In XML config file

- <context:component-scan basePackages="\_\_\_"/>

- Annotation based config

- @ComponentScan(basePackages = "pkg")

- includeFilters and excludeFilters can be used to control bean detection.



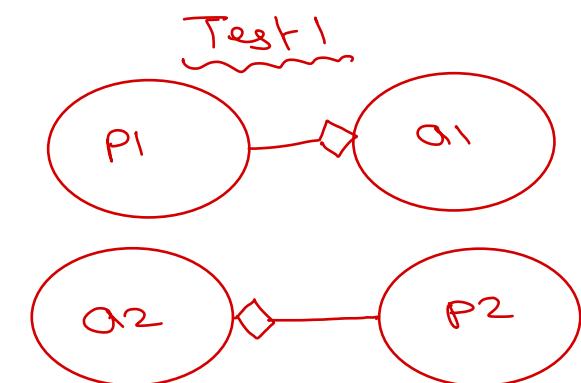
→ @SpringBootApplication  
↳ by default scan current Pkg & sub Pkg.

# Auto-wiring



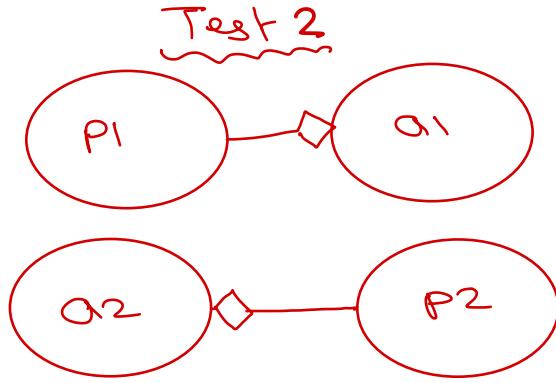
# Auto-wiring

- Automatically injecting appropriate dependency beans into the dependent beans.



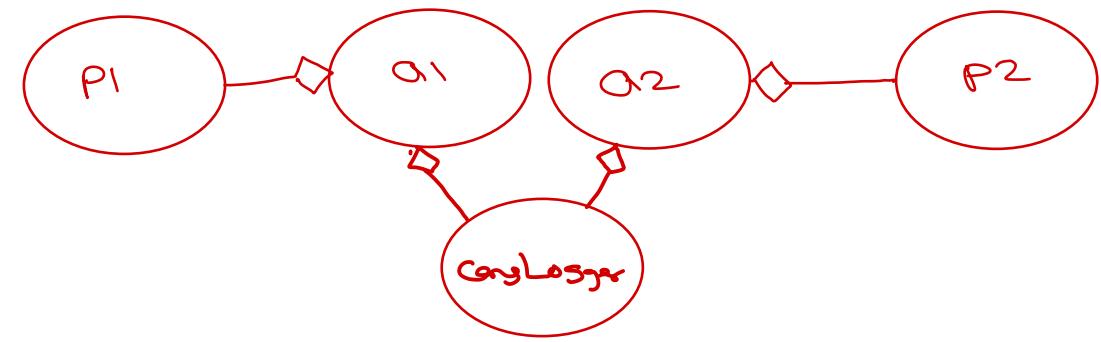
Account Impl  
@Autowired

Logger logger;  
↓  
error: no bean found.



Account Impl  
@Autowired(required=false)

Logger logger;  
no bean found so logger  
is kept null. no error  
raised.



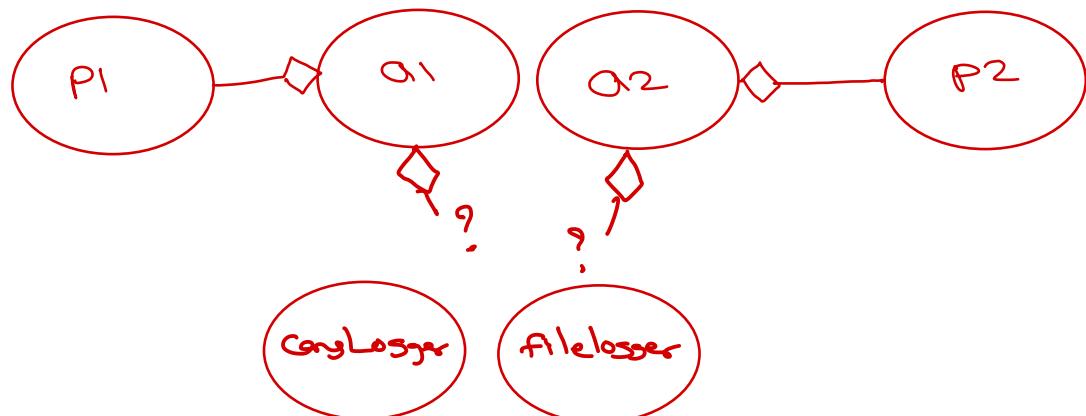
when single bean of logger is available, it is auto wired.

Account Impl  
@Autowired  
Logger logger;

# Auto-wiring

- Automatically injecting appropriate dependency beans into the dependent beans.

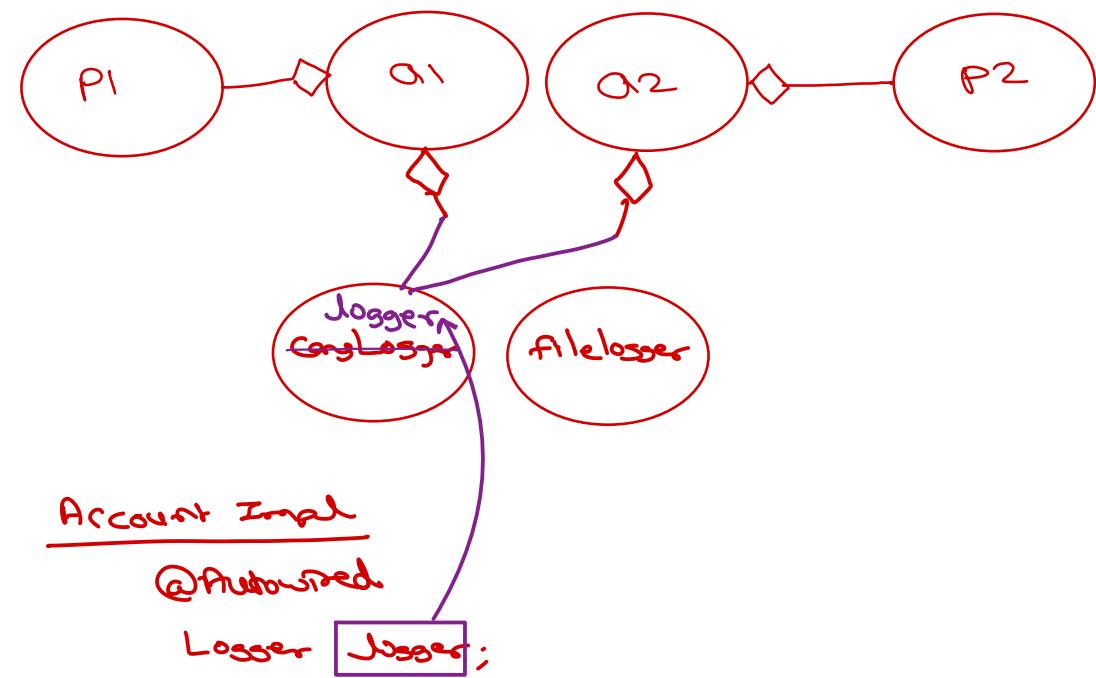
Test 3



Account Impl

@Autowired

Logger logger;



Account Impl

@Autowired

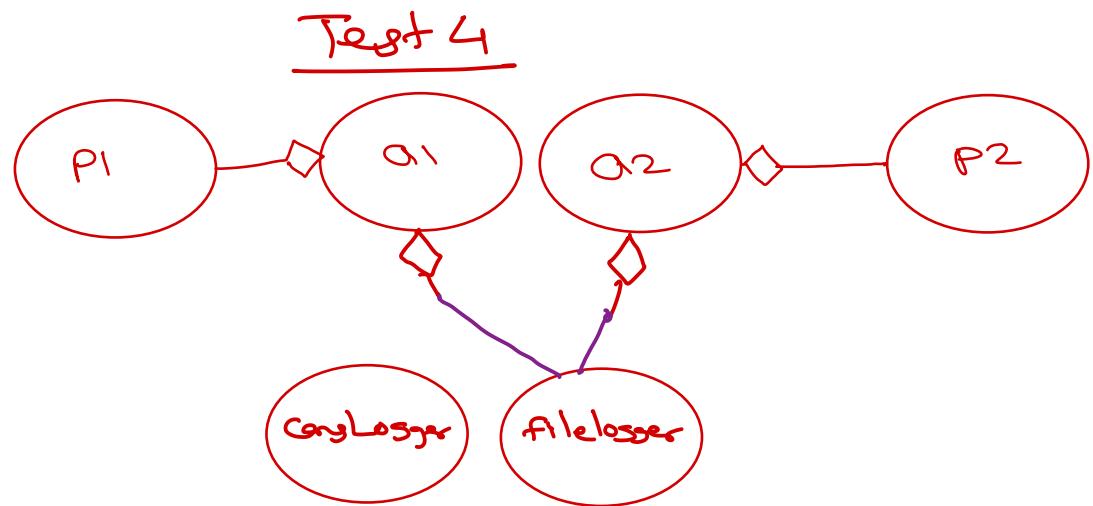
Logger logger;

If multiple beans with same type, it will find bean with name same as fieldname.  
If found, it will auto-wire.



# Auto-wiring

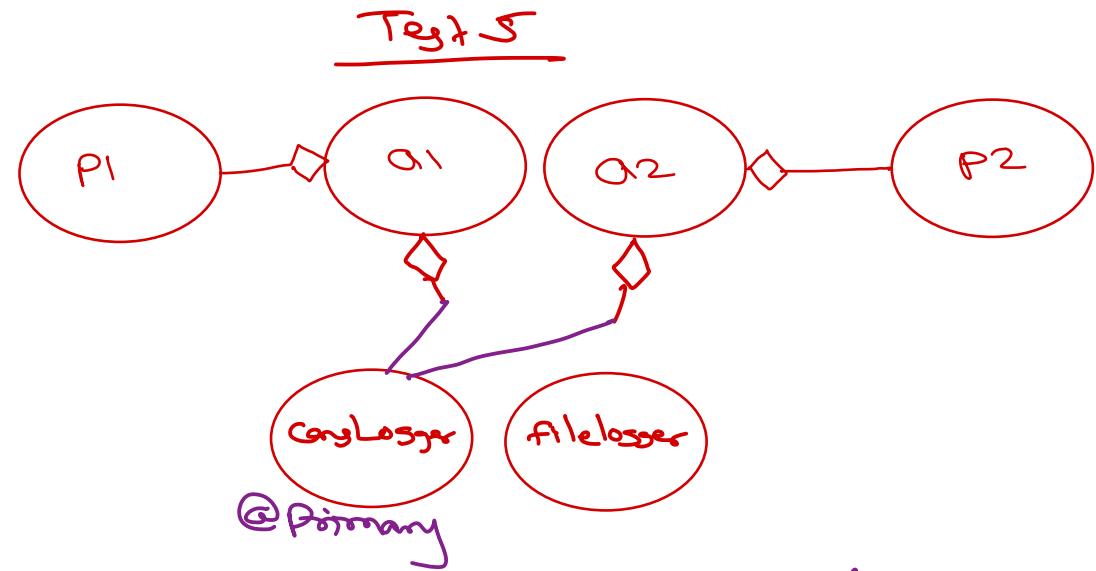
- Automatically injecting appropriate dependency beans into the dependent beans.



## Account Impl

```
@Qualifier("filelogger")
@.Autowired
Logger logger;
```

The ambiguity can be resolved by specifying bean to be autowired using `@Qualifier`.



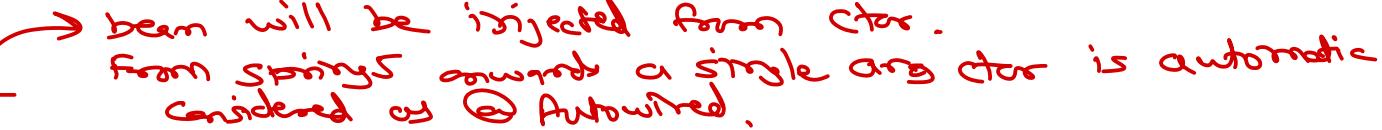
In case of multiple beans of same type, raise importance of one of the bean using `@Primary`. So if conflict, the primary bean will be selected by spring container (instead of error).

## Auto-wiring (XML config)

- `<bean id="..." class="..." autowire="default|no|byType|byName|constructor" .../>`
  - default / no: Auto-wiring is disabled.
  - byType: Dependency bean of property type will be assigned (via setter).
    - If multiple beans of required type are available, then exception is thrown.
    - If no bean of required type is available, auto-wiring is not done.
  - byName: Dependency bean of property name will be assigned (via setter).
    - If no bean of required name is available, auto-wiring is not done.
  - constructor: Dependency bean of property type will be assigned via ~~single argument~~ constructor (of bean type). *default*
- `<bean id="..." class="..." autowire-candidate="true|false" .../>`
  - false -- do not consider this dependency bean for auto-wiring.



# Auto-wiring (Annotation config)

- @Autowired  
    - Setter level: setter based DI
    - Constructor level: constructor based DI 
    - Field level: field based DI
  - @Autowired: Find bean of corresponding field type and assign it.
    - If no bean is found of given type, it throws exception.
      - @Autowired(required=false): no exception is thrown, auto-wiring skipped.
      - If multiple beans are found of given type, it try to attach bean of same name. and if such bean is not found, then throw exception.
      - If multiple beans are found of given type, programmer can use @Qualifier to choose expected bean. @Qualifier can only be used to resolve conflict in case of @Autowired.
  - @Resource (JSR 250) 
    - @Resource: DI byName, byType, byQualifier   
  - @Inject (JSR 330) – same as @Autowired
    - @Inject: DI byType, byQualifier (@Named), byName   
-   
@Autowired  
by Type  
by Qualifier  
by Name  
error.



# Mixed configuration

- **XML config + Annotations**

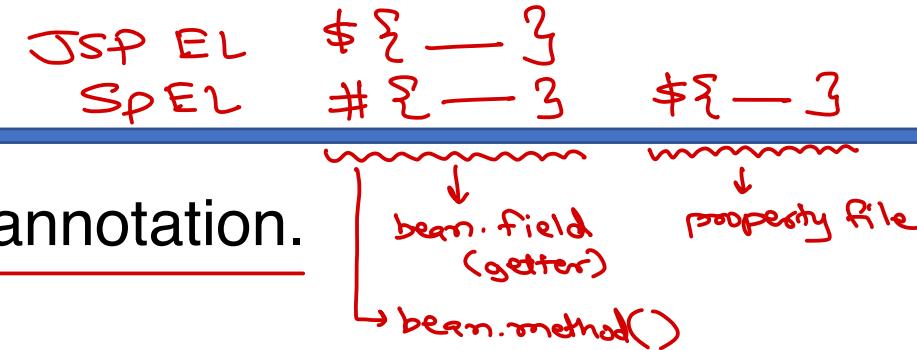
- XML config file is loaded using ClassPathXmlApplicationContext.
- <context:annotation-config/> activate annotation processing.
  - @PostConstruct
  - @PreDestroy
  - @Autowired
  - @Qualifier



# Spring EL



# Spring Expression Language



- SPEL can be used with XML config or @Value annotation.
- Using XML config
  - value="#{bean.field}"
  - value="\${property-key}"
    - In beans.xml: <context:property-placeholder location="classpath:app.properties" />
- Using Annotation config
  - Auto-wire dependency beans: #{bean}
  - Values using SPEL expressions: #{bean.field} or #{bean.method()}
  - Values using SPEL expressions from properties files: \${property-key}
    - In @Configuration class: @PropertySource("classpath:app.properties")

( if using annotation config / spring boot

# Spring Expression Language

- SpEL is a powerful expression language that supports querying and manipulating an object graph at runtime. Syntactically it is similar to EL.<sup>(JSP)</sup>
- SpEL can be used in all spring framework components/products.
- SpEL supports Literal expressions, Regular expressions., Class expressions, Accessing properties, Collections, Method invocation, Relational operators, Assignment, Bean references, Inline lists/maps, Ternary operator, etc.
- SpEL expressions are internally evaluated using SpELExpressionParser.
  - ExpressionParser parser = new SpELExpressionParser();
  - value = parser.parseExpression("'Hello World'.concat('!')");
  - value = parser.parseExpression("new String('Hello World').toUpperCase()");
  - value = parser.parseExpression("bean.list[0]");
- SpEL expressions are slower in execution due parsing. Spring 4.1 added SpEL compiler to speed-up execution by creating a class for expression behaviour at runtime.



### singleton class



only one object of singleton class is created.

if tried to create next obj, get ref of already created object.

### spring bean



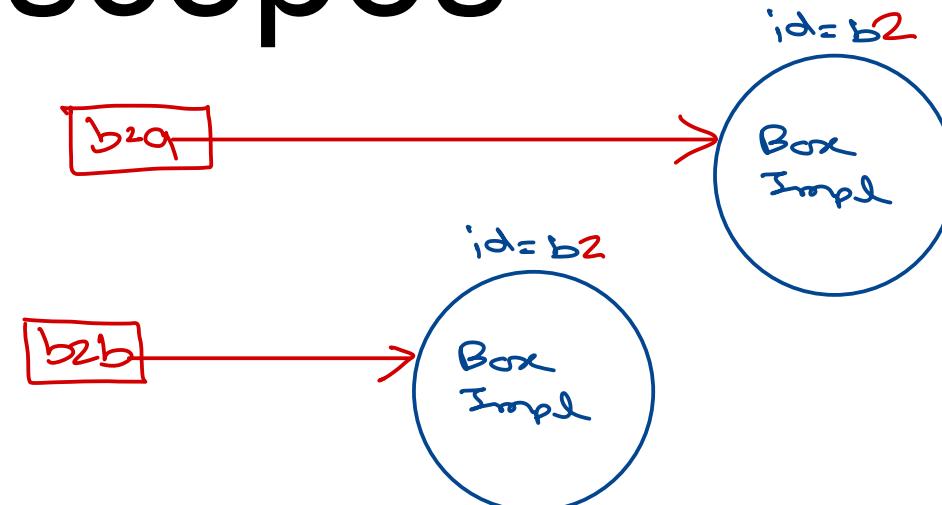
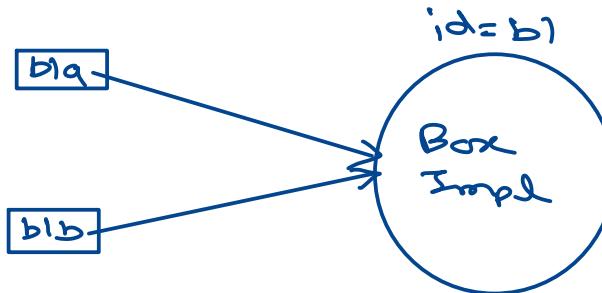
object of a java class created by Spring container.

### singleton bean



- ✓ only one bean object is created (of given id).
- ✓ typically when appctx is created.
- ✓ each access to bean return same obj.
- ✓ managed/stored in Spring Container.
- ✓ destroyed when Spring Container shutdown.

# Spring bean scopes



# Bean scopes

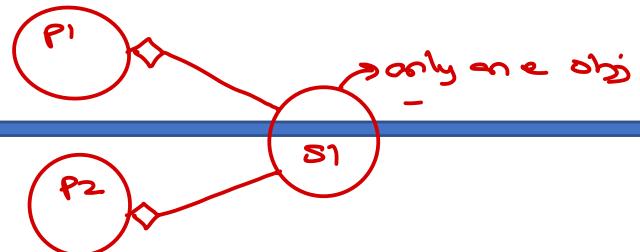
- Bean scope can be set in XML or annotation.
  - <bean id="\_\_\_" class="\_\_\_" scope="singleton|prototype|request|session" />
  - @Scope("singleton|prototype|request|session")
- singleton
  - Single bean object is created and accessed throughout the application.
  - XMLBeanFactory creates object when getBean() is called for first time for that bean.
  - ApplicationContext creates object when ApplicationContext is created.
  - For each subsequent call to getBean() returns same object reference.
  - Reference of all singleton beans is managed by spring container.
  - During shutdown, all singleton beans are destroyed (@PreDestroy will be called).
- prototype
  - No bean is created during startup.
  - Reference of bean is not maintained by ApplicationContext.
  - Beans are not destroyed automatically during shutdown.
  - Bean object is created each time getBean() is called.
- request and session: scope limited to current request and session.



# Bean scopes



# Bean scopes

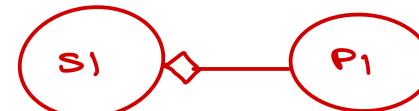


- Singleton bean inside prototype bean

- Single singleton bean object is created.
- Each call to getBean() create new prototype bean. But same singleton bean is autowired with them.

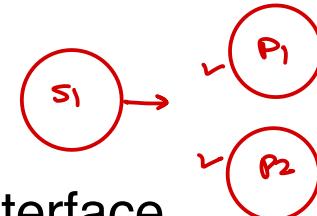
- Prototype bean inside singleton bean

- Single singleton bean object is created.
- While auto-wiring singleton bean, prototype bean is created and is injected in singleton bean.
- Since there is single singleton bean, there is a single prototype bean.



- Need multiple prototype beans from singleton bean (solution 1)

- Using `ApplicationContextAware`
- The singleton bean class can be inherited from `ApplicationContextAware` interface.
- When its object is created, container call its `setApplicationContext()` method and give current `ApplicationContext` object. This object can be used to create new prototype bean each time (as per requirement).



- Need multiple prototype beans from singleton bean (solution 2)

- using `@Lookup` method
- The singleton bean class contains method returning prototype bean.
- If method is annotated with `@Lookup`, each call to the method will internally call `ctx.getBean()`. Hence for prototype beans, it returns new bean each time.



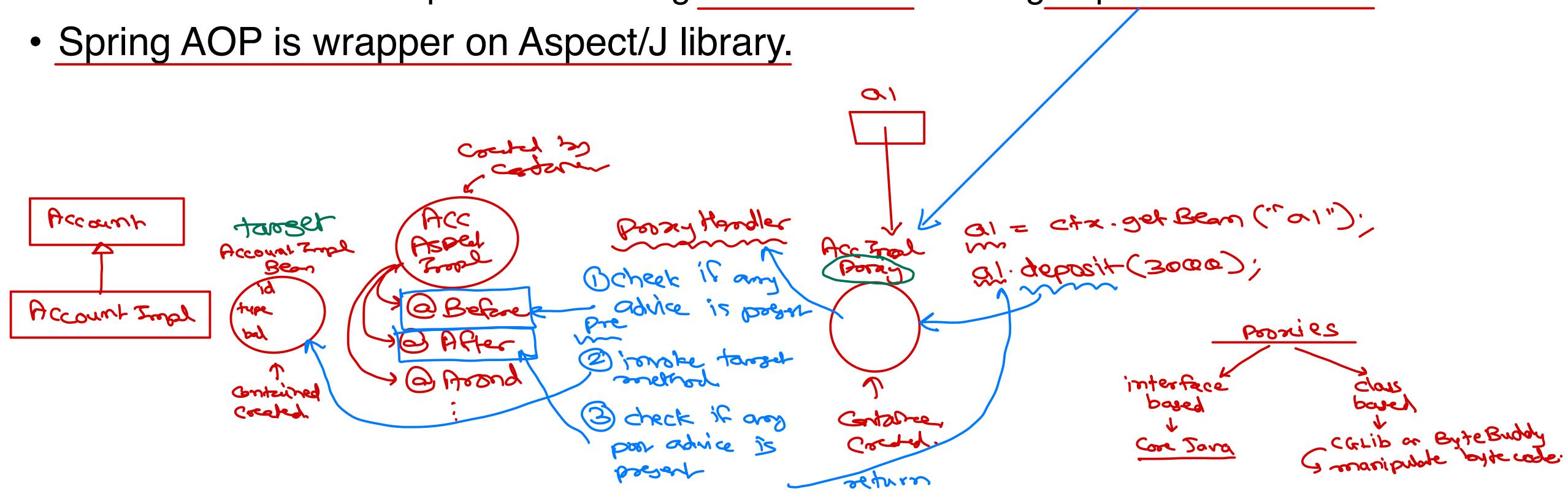
# Spring AOP



# Spring AOP

→ monitor, log, profile, tx, security -

- Implementation of cross cutting concerns without modifying core business logic.
  - Pre-processing & Post-processing
  - In Java EE it is implemented using Filters.
  - In Java it can be implemented using Java Proxies or using Aspect/J framework.
- Spring AOP is wrapper on Aspect/J library.



# Spring AOP

- Aspect: Implementation of cross-cutting concerns.
  - In Spring AOP, it is a class containing cross-cutting concern implementation - @Aspect.
  - It will contain implementation of one or more advices implementations.
- Advice: Implementation of a pre-processing or post-processing or both.
  - In Spring AOP, it is a method in @Aspect class.
  - Types: @Before, @After, @Around, @AfterThrowing , @AfterReturning  
Pre Post Pre-Post Post After Returning Post
- JoinPoint: Point of intersection where advice is applied.
  - In Spring AOP, it is an object containing info about the method on which advice is applied and the object on which that method is invoked.
  - It is argument to the advice methods i.e. JoinPoint or ProceedingJoinPoint.
- Pointcut: Combination of multiple JoinPoints.
- Target: The object on which business logic method is invoked.
- Proxy: The wrapper on the target object.
  - Responsible for executing advices & actual business logic.
  - Proxy can be interface based or cglib.



# Spring AOP

- Add AOP dependencies in pom.xml i.e. spring-aop, aspectjweaver
- Implement aspect class and advices in it.
  - @Before("execution (\* Account.get(..))")
  - @After("execution (\* Account.set(..))")
  - @Pointcut("execution (\* Account.withdraw(..)) || execution (\* Account.deposit(..))")
  - @Around("transaction()")
- For XML (mixed) config, in bean config file
  - <aop:aspectj-autoproxy>
- For Annotation config, in config class
  - @EnableAspectJAutoProxy



# Spring JDBC



# JDBC Quick Revision

- JDBC is specification given by Sun/Oracle.
- Specification interfaces are implemented by driver.
  - Driver, Connection, Statement, ResultSet
- JDBC driver convert Java request to RDBMS understandable form and RDBMS response to Java understandable form.
- JDBC programming steps
  - Add JDBC driver into project CLASSPATH.
  - Load and register JDBC driver.
  - Create JDBC connection.
  - Prepare JDBC statement.
  - Execute query and process result.
  - Close all.



# JDBC Quick Revision

- Transaction is a set of DML queries executed as a single unit. If any query fails, other queries are discarded.
- Transaction is feature of RDBMS.
- RDBMS commands: START TRANSACTION, SAVEPOINT, COMMIT, ROLLBACK.
- It follows ACID properties: Atomicity, Consistency, Isolation and Durability.
- JDBC functions
  - `con.setAutocommit(false);`
  - `con.commit();`
  - `con.rollback();`

```
try {  
    con.setAutocommit(false);  
    // exec DML statements  
    con.commit();  
} catch(Exception e) {  
    con.rollback();  
}
```



# Spring JDBC Integration

- Spring DI simplifies JDBC programming.
- Using JDBC we can avoid overheads of ORM tools. This is helpful in small applications, report generation tools or running ad-hoc SQL queries.
- Steps:
  - In pom.xml, add spring-jdbc and mysql-connector-java
  - Create dataSource bean (XML or annotation config)
  - Create JdbcTemplate bean (XML or annotation config) and attach dataSource,
  - Implement RowMapper interface in a class (for dealing with SELECT queries)
    - mapRow() convert resultset row to Java object.
  - Create Spring @Repository bean and auto-wire JdbcTemplate in it.
  - Invoke JdbcTemplate query() and/or update() for appropriate operations.
  - To use transaction management, create TransactionManager bean and use @Transactional on service layer (common practice).



# Spring JDBC Integration

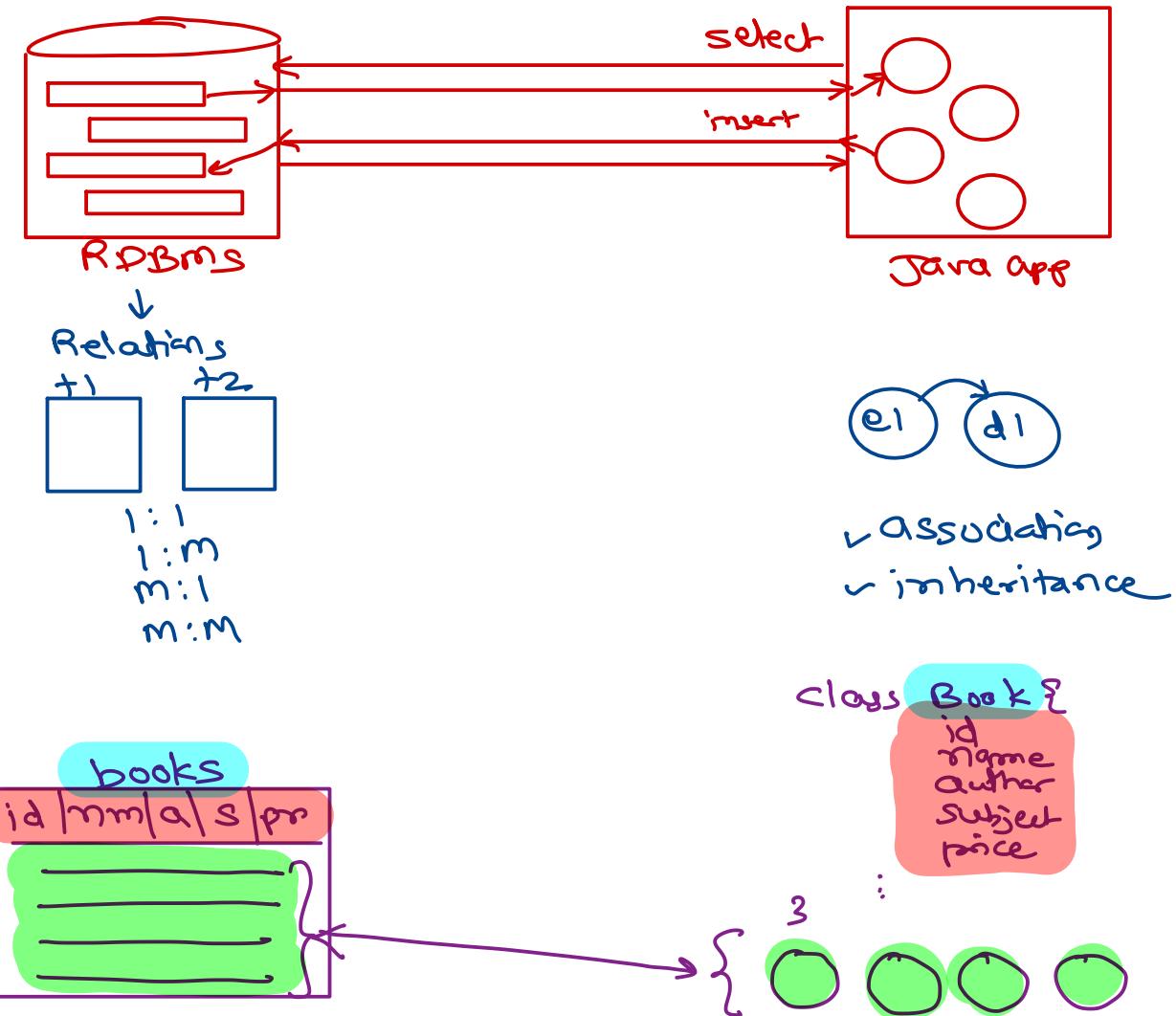


# Hibernate

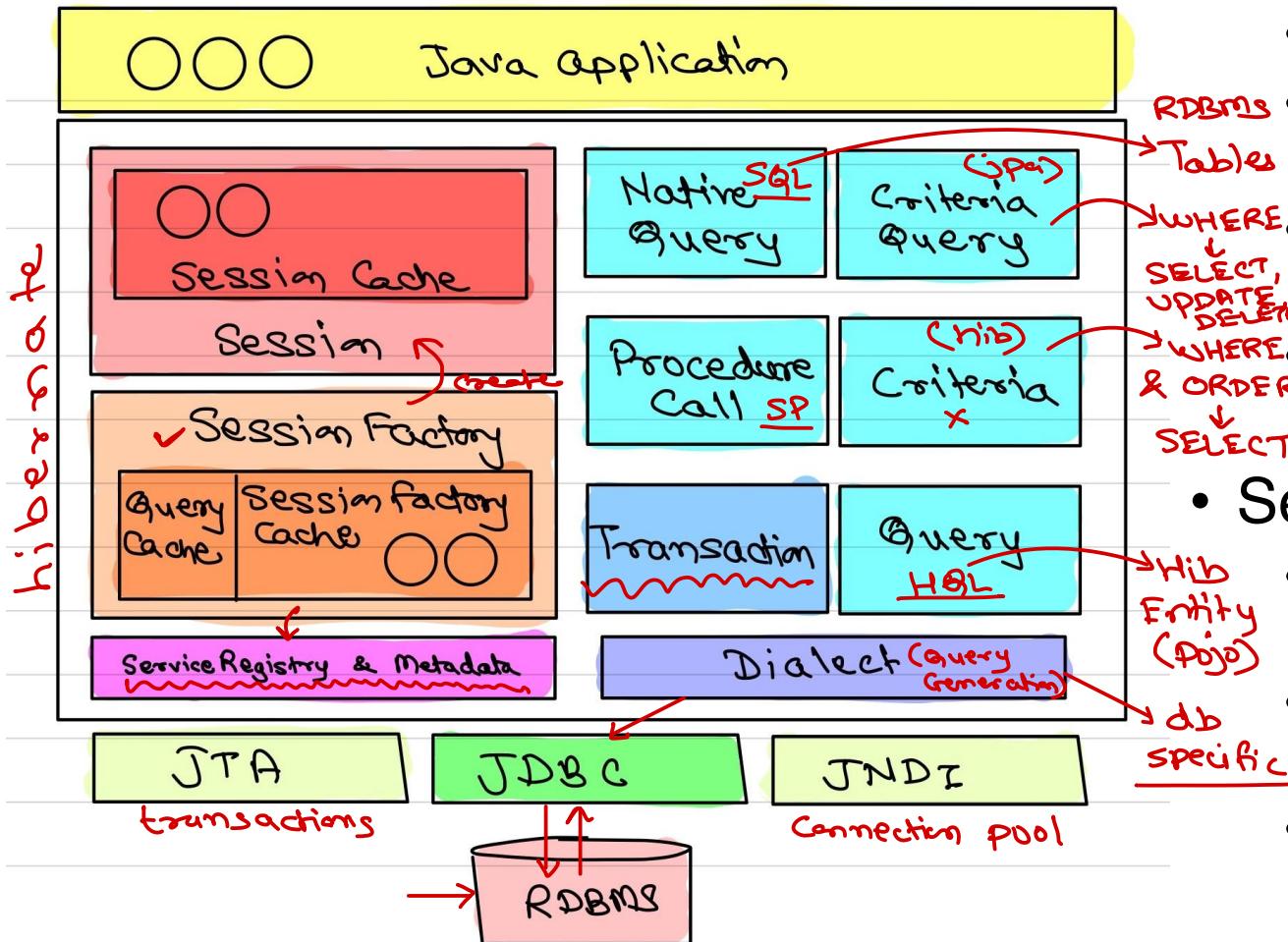


# Object Relational Mapping

- Converting Java objects into RDBMS rows and vice-versa is done manually in JDBC code.
- This can be automated using Object Relational Mapping.
- Class → Table and Field → Column
- It also map table relations into entities associations/inheritance and auto-generates SQL queries.
- Hibernate is most popular ORM tool.
- Other popular ORM are EclipseLink, iBatis, Torque, ...
- JPA is specification for ORM.  
↳ Java Persistence API → Sun/Oracle.



# Hibernate



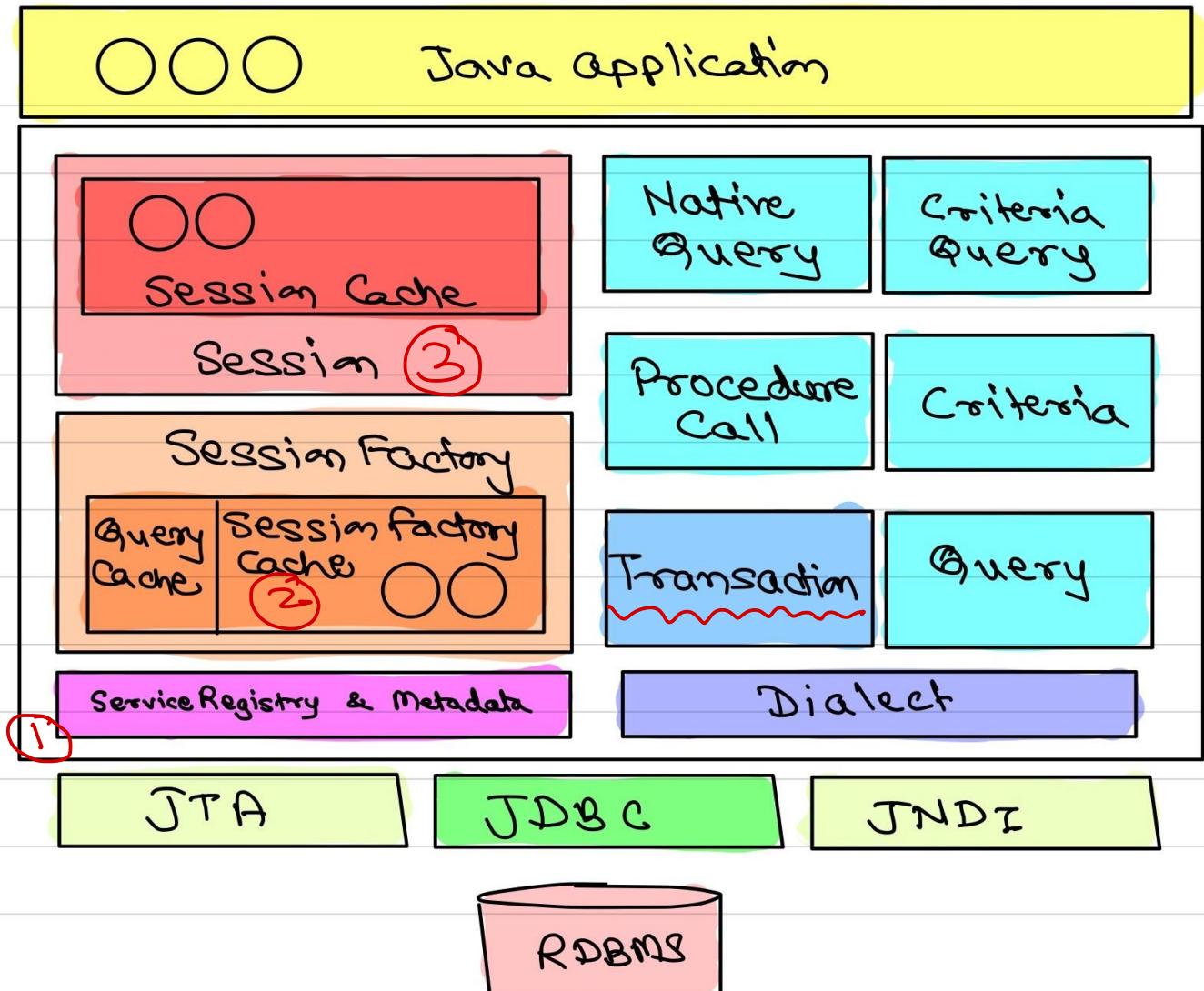
## • SessionFactory

- One SessionFactory per application (per db).
- Heavy-weight object. Not recommended to create multiple instances.
- Thread-safe. Can be accessed from multiple threads (synchronization is built-in).
- Typical practice is to create singleton utility class for that.

## • Session

- Created by SessionFactory & it encapsulates JDBC connection.
- All hibernate operations are done on hibernate sessions.
- Is not thread-safe. Should not access same session from multiple threads.
- Light-weight. Can be created and destroyed as per need.

# Hibernate



- Transaction

- In hibernate, autocommit is false by default.
- DML operations should be performed using transaction.
- session.beginTransaction()
  - to start new transaction.
- tx.commit()
  - to commit transaction.
- tx.rollback()
  - to rollback transaction.

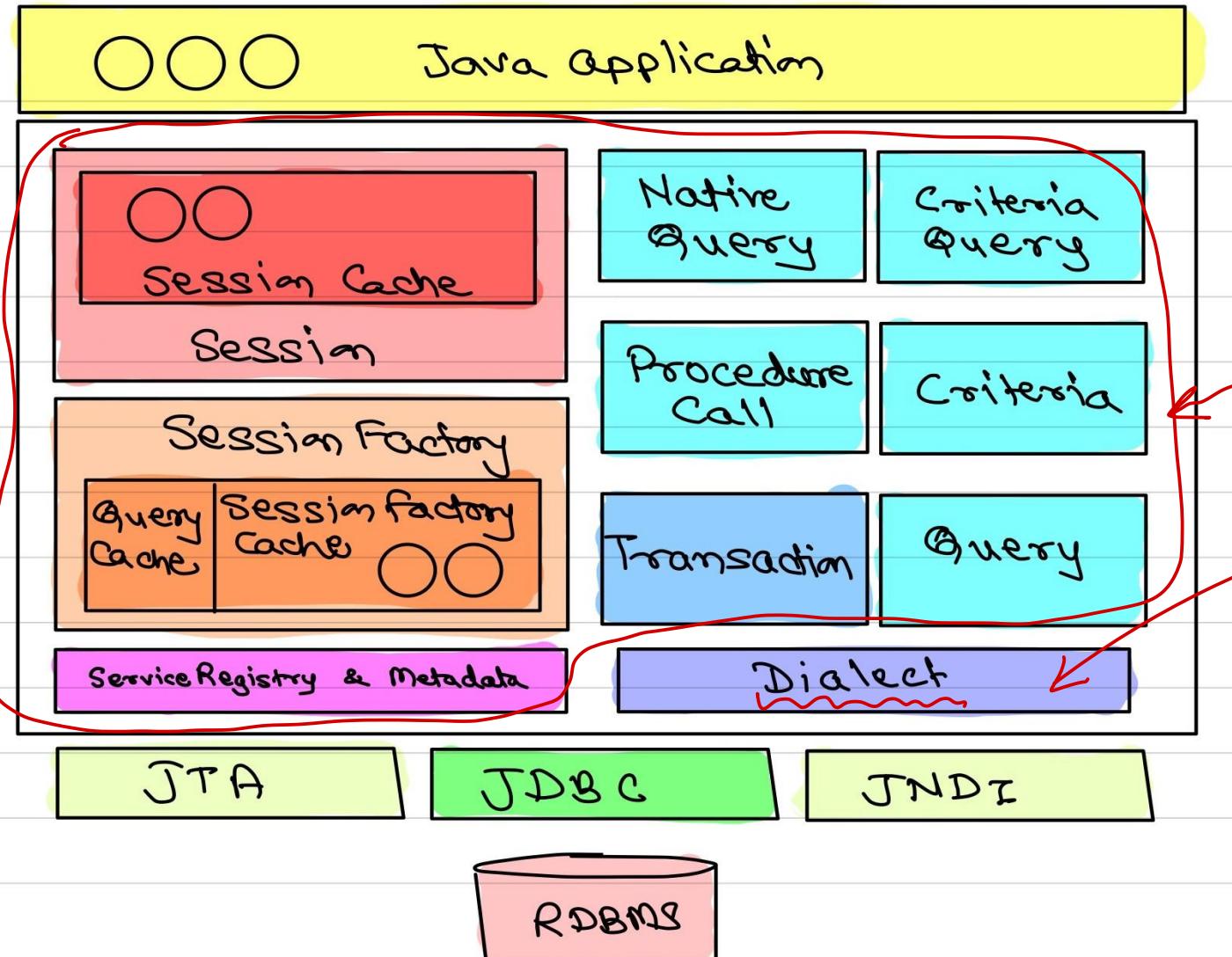
```
try {  
    // begin tx  
    tx = session.beginTransaction();  
  
    // Commit  
    tx.commit();  
  
} catch (...) {  
    // Rollback  
    tx.rollback();  
}
```

- Hibernate CRUD methods

- get() → select
- save() → insert
- update() → update
- delete() → delete

*session* } *transaction*

# Hibernate



## Dialect

- RDBMS have specific features like data types, stored procedures, primary key generation, etc. *Popular*
- Hibernate support all RDBMS.
- Most of code base of Hibernate is common.
- Database level changes are to be handled specifically and appropriate queries should be generated. This is handled by Dialect.
- Hibernate have dialects for all RDBMS. Programmer should configure appropriate dialect to utilize full features of RDBMS.

# Hibernate 3 Bootstrapping

```
public class HbUtil {  
    private static SessionFactory factory;  
  
    static {  
        try {  
            ① Configuration cfg = new Configuration();  
            ② cfg.configure(); load hibernate.cfg.xml  
            ③ factory = cfg.buildSessionFactory();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    public static SessionFactory getSessionFactory() {  
        return factory; wrapping  
    }  
  
    public static void shutdown() {  
        factory.close();  
    }  
}
```

*empty obj*

*as per Config*

- Hibernate Configuration (*hibernate-  
cfg.xml*)
  - hibernate.connection.driver\_class
  - hibernate.connection.url
  - hibernate.connection.username
  - hibernate.connection.password
  - hibernate.dialect
  - hibernate.show\_sql
- Hibernate3 Bootstrapping
  - ① Create Configuration object.
  - ② Read hibernate.cfg.xml file using its configure() method.
  - ③ Create SessionFactory using its buildSessionFactory() method.



# Hibernate 5 Bootstrapping

```
public class HbUtil {  
    private static final SessionFactory factory  
        = createSessionFactory();  
    private static ServiceRegistry serviceRegistry;  
  
    private static SessionFactory createSessionFactory() {  
        ① serviceRegistry = new StandardServiceRegistryBuilder()  
            .configure() // read from hibernate.cfg.xml  
            .build();  
  
        ② Metadata metadata = new MetadataSources(serviceRegistry)  
            .getMetadataBuilder()  
            .build();  
  
        ③ return metadata.getSessionFactoryBuilder().build();  
    }  
    public static void shutdown() {  
        factory.close();  
    }  
    public static SessionFactory getSessionFactory() {  
        return factory;  
    }  
}
```

## Hibernate 5 Bootstrapping

- ① Create ServiceRegistry.
- ② Create Metadata.
- ③ Create SessionFactory.

### ServiceRegistry

- ServiceRegistry is interface.
- Some implementations are
  - ✓ StandardServiceRegistry,
  - ✓ BootstrapServiceRegistry,
  - ✓ EventListenerRegistry, ...
- Add, manage hibernate services.

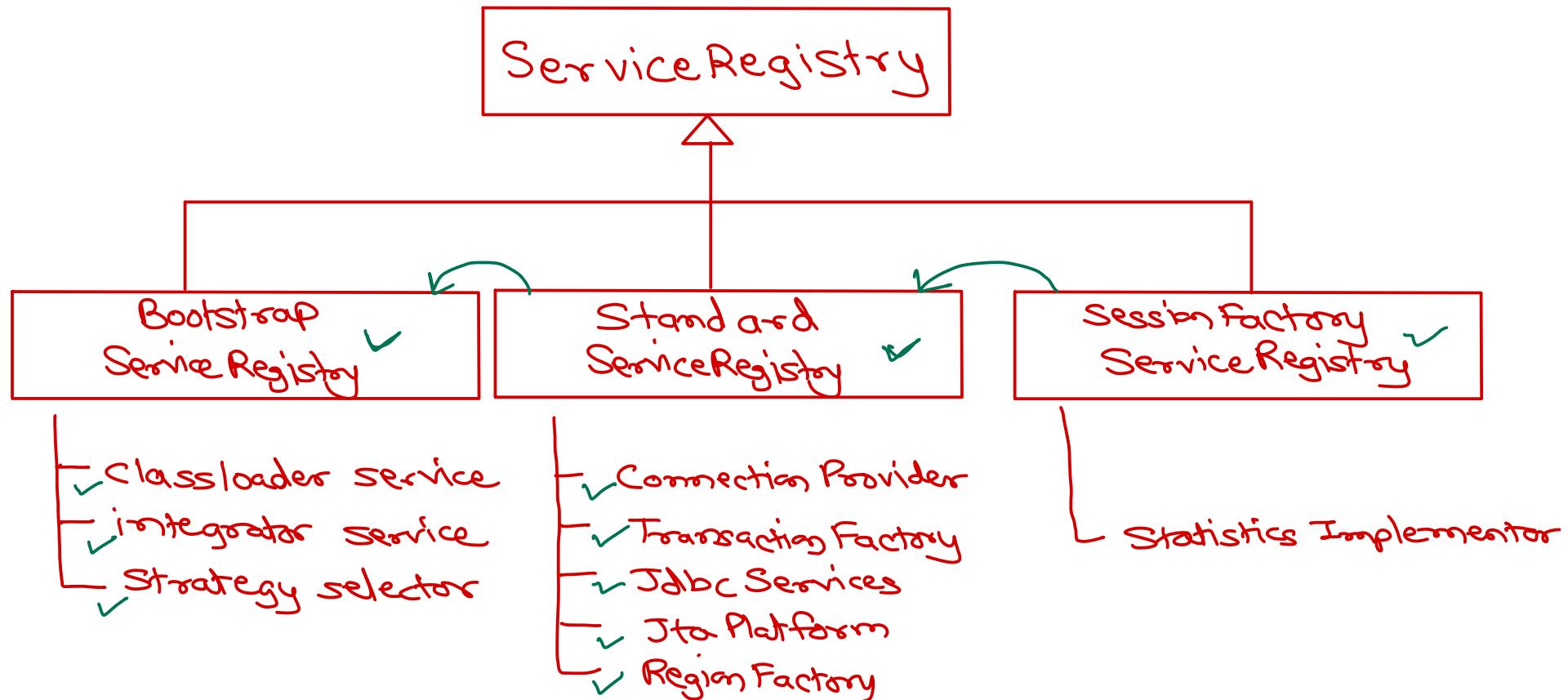
### Metadata

- Represents application's domain model & its database mapping. → ORM

✓ <https://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html>



# Hibernate service registry



# Hibernate

- Hibernate3 added annotations for ORM.

- ORM using annotations

- @Entity
- @Table
- @Column
- @Id
- @Temporal
- @Transient

→ to convert date & time  
classes from java.sql  
to java.util.Date and/or  
java.util.Calendar.

Can be used to map  
Temporal.DATE → mysql  
DATE  
Temporal.TIME → mysql  
TIME  
Temporal.TIMESTAMP → mysql  
DATETIME  
TIMESTAMP

- @Column can be used on field level or on getter methods.



# Hibernate

hibernate.cfg.xml → <mapping> — />

- Earlier versions does ORM using .hbm.xml files.

```
@Entity  
@Table(name = "DEPT")  
public class Dept implements Serializable {  
    @Id //primary key  
    @Column(name = "deptno")  
    private int id; //deptno  
    @Column(name = "dname")  
    private String name; //dname  
    @Column(name = "loc")  
    private String loc; //loc  
    ...  
}
```

annotations

```
<hibernate-mapping>  
  <class name="com.sunbeam.sh.Dept" table="DEPT">  
    <id name="id" type="int">  
      <column name="DEPTNO" />  
      <generator class="assigned" />  
    </id>  
    <property name="name" type="java.lang.String">  
      <column name="DNAME" />  
    </property>  
    <property name="loc" type="java.lang.String">  
      <column name="LOC" />  
    </property>  
  </class>  
</hibernate-mapping>
```



# Hibernate Transaction

- In hibernate, autocommit is false by default.
- DML operations should be performed using transaction.
  - session.beginTransaction(): to start new tx.
  - tx.commit() & tx.rollback(): to commit/rollback tx.
- session.flush()
  - Forcibly synchronize in-memory state of hibernate session with database.
  - Each tx.commit() automatically flush the state of session.
  - Manually calling flush() will result in executing appropriate SQL queries into database.
  - Note that flush() will not commit the data into the RDBMS tables.
  - The flush mode can be set using session.setHibernateFlushMode(mode).
    - ALWAYS, AUTO, COMMIT, MANUAL
- If hibernate.connection.autocommit is set to true, we can use flush to force executing DML queries.



# openSession() vs getCurrentSession()

- openSession()
  - Create new hibernate session.
  - It is associated with JDBC connection (autocommit=false).
  - Can be used for DQL (get records), but cannot be used for DML without transaction.
  - Should be closed after its use.
- getCurrentSession()
  - Returns session associated with current context.
  - Session will be associated with one of the context (hibernate.current\_session\_context\_class).
    - thread: Session is stored in TLS.
    - jta: Session is stored in transaction-context given by JTA providers (like app servers).
    - custom: User implemented context.
  - This session is not attached with any JDBC connection.
  - JDBC connection is associated with it, when a transaction is created. The connection is given up, when transaction is completed.
  - The session is automatically closed, when scope is finished. It should not be closed manually.

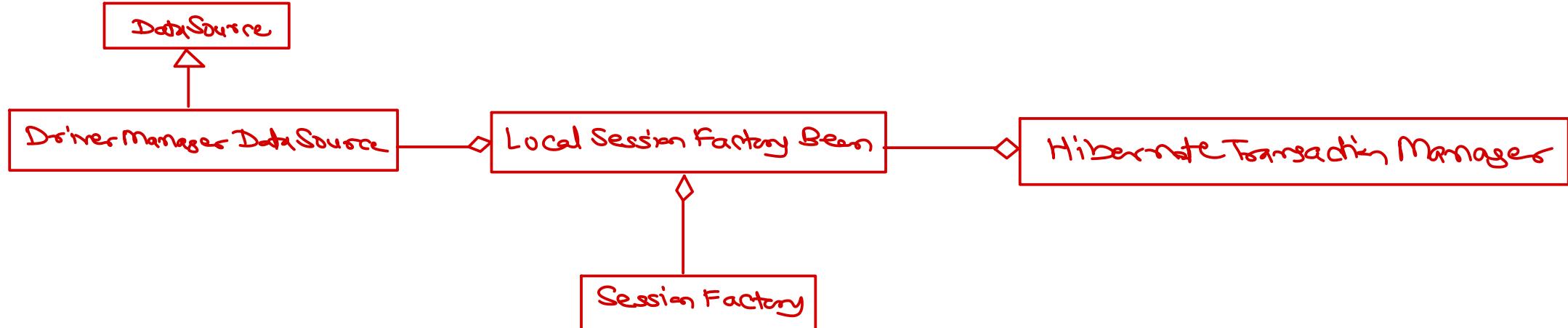


# Using hibernate in Java EE application

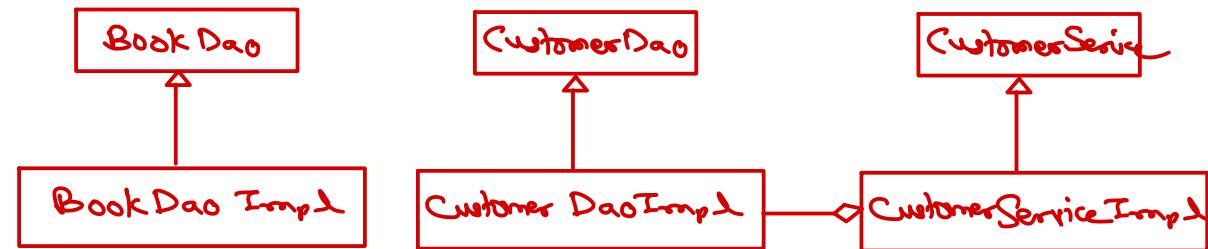
- While dealing with single database, whole Java EE application should have single SessionFactory instance.
- This can be created as Singleton object in ServletContext. *(or as a singleton class HbUtil).*
- For each request new thread gets created.
- Hibernate session can be attached to thread session context, so that it can be accessed from any component.
- ORM is done using annotations on Entity objects and DAO layer is to be written using Hibernate (instead of JDBC).

Refer steps from notes.





# Spring Hibernate



# Spring Hibernate Integration *(Spring core)*

- Spring DI simplifies Hibernate ORM.
- LocalSessionFactoryBean provides session factory, while transaction automation is done by HibernateTransactionManager bean.
- Steps:
  - In pom.xml, add spring-orm, mysql-connector-java and hibernate-core.
  - Create dataSource, sessionFactory (with hibernate config), transactionManager beans. Also set default transactionManager. PackagesToScan ↗
  - Implement entity classes. Ensure that spring session factory config scan them.
  - Implement @Repository class and auto-wire session factory. Use factory.getCurrentSession() to obtain hibernate session and perform operations.
  - Implement @Service layer and mark business logic methods as @Transactional.
    - Note that single business operation (from service layer) may deal with multiple operations on different repositories. @Transactional put all ops under same tx.

*@Transactional can be used on @Repository. However using on @Service is recommended*



# Spring Boot Hibernate Integration

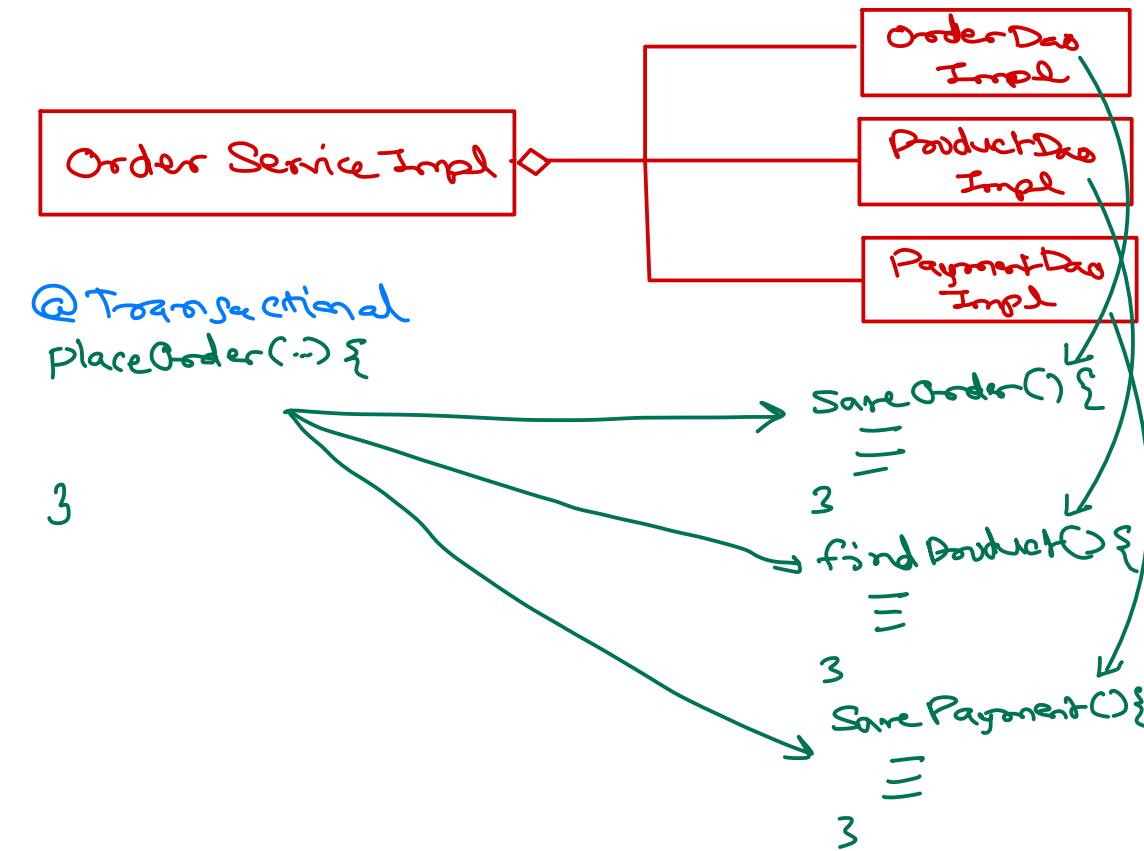
- Spring Boot Auto-configuration & DI simplifies Hibernate ORM.
- LocalSessionFactoryBean provides session factory, while transaction automation is done by HibernateTransactionManager bean.
- Steps:
  - Create spring boot project with Spring Data JPA and MySQL dependencies.
  - Define hibernate & database configurations into application.properties. *(or separate database.properties)*
  - Create a @Configuration class to create dataSource, sessionFactory and transactionManager beans. Mark it with @EnableTransactionManagement.
  - Implement entity classes with appropriate annotations.
  - Implement @Repository class and auto-wire session factory. Use factory.getCurrentSession() to obtain hibernate session and perform operations.
  - Implement @Service layer and mark business logic methods as @Transactional.
    - Note that single business operation (from service layer) may deal with multiple operations on different repositories. @Transactional put all ops under same tx.

*@Transactional can be used on @Repository. However using on @Service is recommended.*



# @Service layer

- @Repository layer contains database operations (i.e. CRUD operations, ...).
- @Service layer contains business logic. It is implemented as per business operations.
- One @Service component may have one or more DAO component dependencies.
- It is common practice to handle transactions in service layer.



# @Transactional



- @Transactional is declarative transaction management of Spring.
- It can be used method level or class level.  
If used on class level, it applies to all methods in the class.
- Spring internally use JDBC transaction in AOP fashion.
  - ✓ start transaction (before method).
  - ✓ commit transaction (if method is successful).
  - ✓ rollback transaction (if method throw exception).
- Transaction management is done by platform transaction manager e.g. datasource, hibernate or jpa transaction manager.



# @Transactional

- If one transactional method invokes another transactional method, transaction behaviour is defined by propagation attribute.

- REQUIRED: Support a current transaction, create a new one if none exists. (default)
- REQUIRES\_NEW: Create a new transaction, and suspend the current transaction if one exists.
- SUPPORTS: Support a current transaction, execute non-transactionally if none exists.
- MANDATORY: Support a current transaction, throw an exception if none exists.
- NEVER: Execute non-transactionally, throw an exception if a transaction exists.
- NOT\_SUPPORTED: Execute non-transactionally, suspend the current transaction if one exists.
- NESTED: Execute within a nested transaction (save points) if a current transaction exists, behave like REQUIRED otherwise.

## Customer Service Impl

```
@Transactional  
authenticate(-) {  
    c = findByEmail();  
    ...  
    ...  
    3
```

↓

```
@Transactional  
findByEmail(-) {  
    ...  
    ...  
    3
```

```
@Transactional (isolation = ...)  
    jdbc → Con.setIsolationLevel(...);  
    RDBMS → ACID + x
```

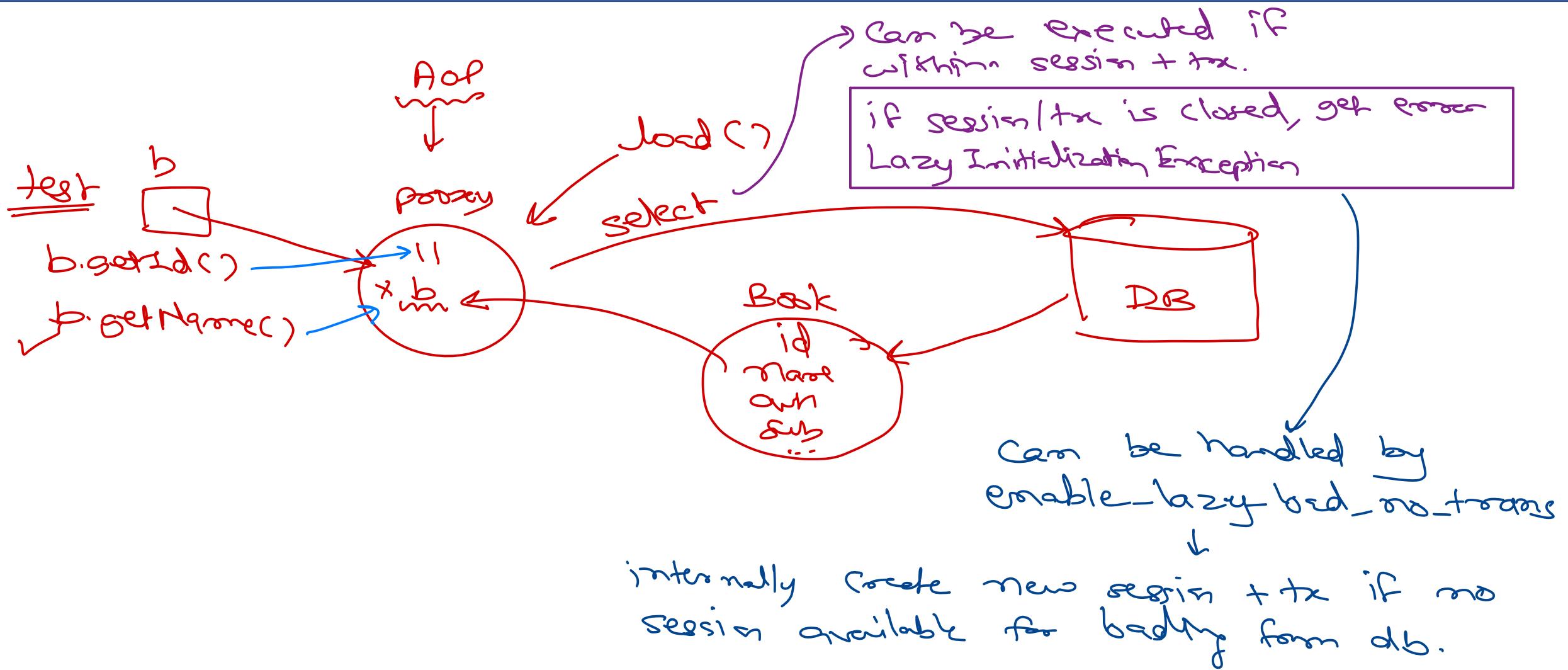


# Hibernate CRUD



- Hibernate Session methods

- get() or find(): Find the database record by primary key and return it. If record is not found, returns null.
- load(): Returns proxy for entity object (storing only primary key). When fields are accessed on proxy, SELECT query is fired on database and record data is fetched. If record not found, exception is thrown.
- save(): Assign primary key to the entity and execute INSERT statement to insert it into database. Return primary key of new record. *return Serializable (PK)*
- persist(): Add entity object into hibernate session. Execute INSERT statement to insert it into database (for all insertable columns) while committing the transaction. *session void*
- update(): Add entity object into hibernate session. Execute UPDATE statement to update it into database while committing the transaction. All (updateable) fields are updated into database (for primary key of entity).
- saveOrUpdate() or merge(): Execute SELECT query to check if record is present in database. If found, execute UPDATE query to update record; otherwise execute INSERT query to insert new record.
- delete() or remove(): Delete entity from database (for primary key of entity) while committing the transaction.
- evict() or detach(): Removes entity from hibernate session. Any changes done into the session after this, will not be automatically updated into the database.
- clear(): Remove all entity objects from hibernate session.
- refresh(): Execute SELECT query to re-fetch latest record data from the database.



# Criteria vs DetachedCriteria

- Criteria
  - represents WHERE clause for SELECT query.
  - cr.add() -- condition and cr.addOrder() -- sort order.
  - for condition -- helper class Restrictions
  - eq(), ne(), gt(), lt(), ...
  - for order -- helper class Order
  - asc(), desc()
  - Created using session.createCriteria().
  - Have methods to fire SELECT query
  - cr.list(), cr.uniqueResult().
- DetachedCriteria
  - Similar to Criteria object, but NOT associated with Session.
  - dcr = DetachedCriteria.forClass(MyEntity.class);
  - dcr.add() -- condition and dcr.addOrder() -- sort order.
  - To execute, call dcr.getExecutableCriteria(session), that gives Criteria object.
  - The Criteria object can give result using list() or uniqueResult().



# Hibernate - Primary keys



# Auto-generate Primary keys

- @GeneratedValue annotation is used to auto-generate primary key.
- This annotation is used with @Id column.
- There are different strategies for generating ids.
  - AUTO: Depends on database dialect.
    - MySQL 8: id will be taken from "next\_val" column of "gen" table.
  - IDENTITY: RDBMS AUTO\_INCREMENT / IDENTITY ← MS-SQL Server
    - MySQL 8: id will be AUTO\_INCREMENT
  - TABLE: Dedicated table for PK generation of all entities
    - MySQL 8: @TableGenerator(name="gen", initialValue=1000, pkColumnName="book\_ids", valueColumnName="id", table="id\_gen", allocationSize=1)
  - SEQUENCE: RDBMS sequence using @SequenceGenerator
    - MySQL 8: Emulated with table.



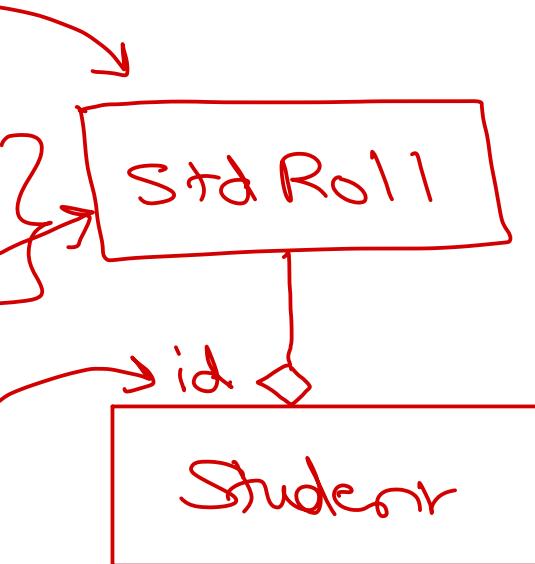
# Auto-generate Primary keys

- `@GenericGenerator` is used to specify config of hibernate's id generation strategies.
- Hibernate allows few more strategies.
  - native: depends on db/dialect.
  - guid: 128-bit unique id
  - hilo: id gen using high low algo.
  - identity: auto incr or identity col.
  - sequence: db seq.
  - increment: select  $\text{max}(\text{id})+1$  from table.
  - foreign: id corresponding to foreign key `@PrimaryKeyJoinColumn`.



# Composite Primary key

- Hibernate @Id is always **serializable**.
- In case of composite PK, create a new class for PK and mark as @Embeddable.
- Create object of that class into entity class & mark it as @EmbeddedId.
- @Embeddable class must implement equals() and hashCode().

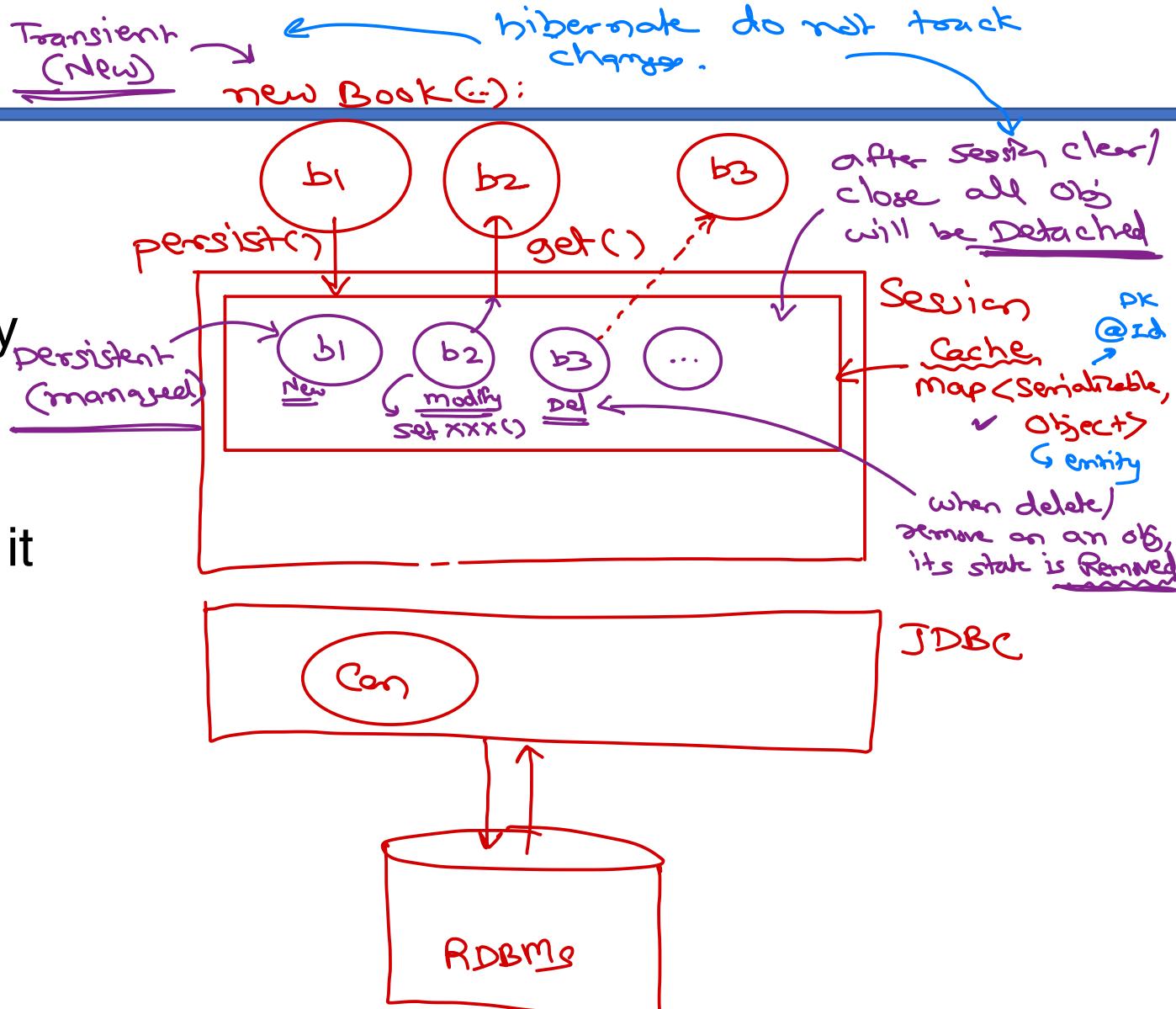


# Hibernate Entity life cycle



# Hibernate session cache

- Collection of entities per session – persistent objects.
- Hibernate keep track of state of entity objects and update into database.
- Session cache cannot be disabled.
- If object is present in session cache, it is not searched into session factory cache or database.
- Use refresh() to re-select data from the database forcibly.



# Hibernate – Entity life cycle

- Transient

- New Java object of entity class.
- This object is not yet associated with hibernate.

- Persistent

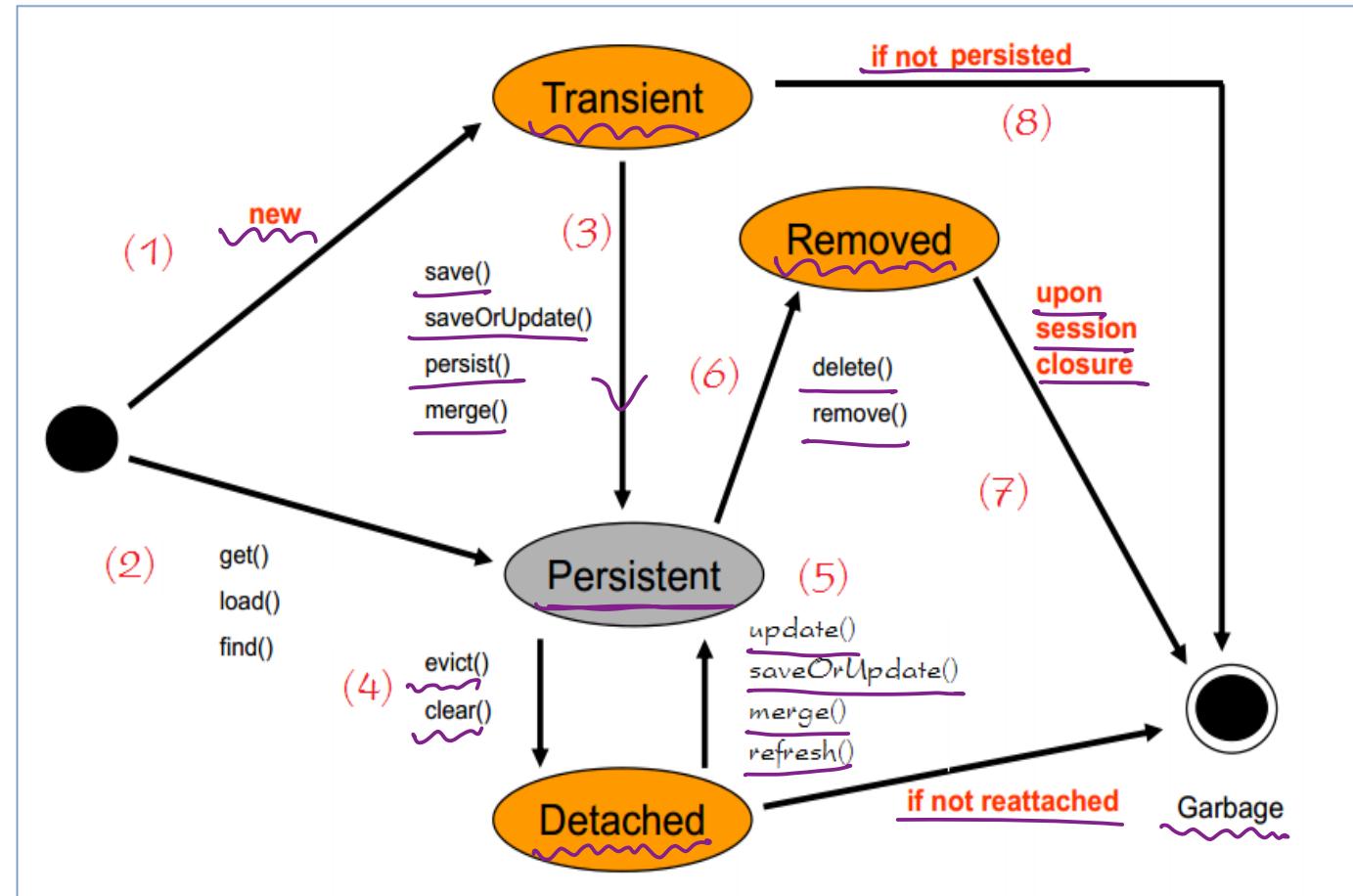
- Object in session cache.
- For all objects created by hibernate or associated with hibernate.
- State is tracked by hibernate and updated in database during commit.

- Never garbage collected.

- Detached
- Object removed from session cache.

- Removed

- Object whose corresponding row is deleted from database.



# Hibernate Entity associations



# Hibernate Relations

- RDBMS Relations

- RDBMS tables are designed by process of Normalization.
- To avoid redundancy, data is organized into multiple tables.
- These tables are related to each other by means of primary key and foreign key constraints.
- ER diagram shows relations among the tables.
- The relations can be:
  - OneToOne ✓
  - OneToMany ✓
  - ManyToOne ✓
  - ManyToMany ✓

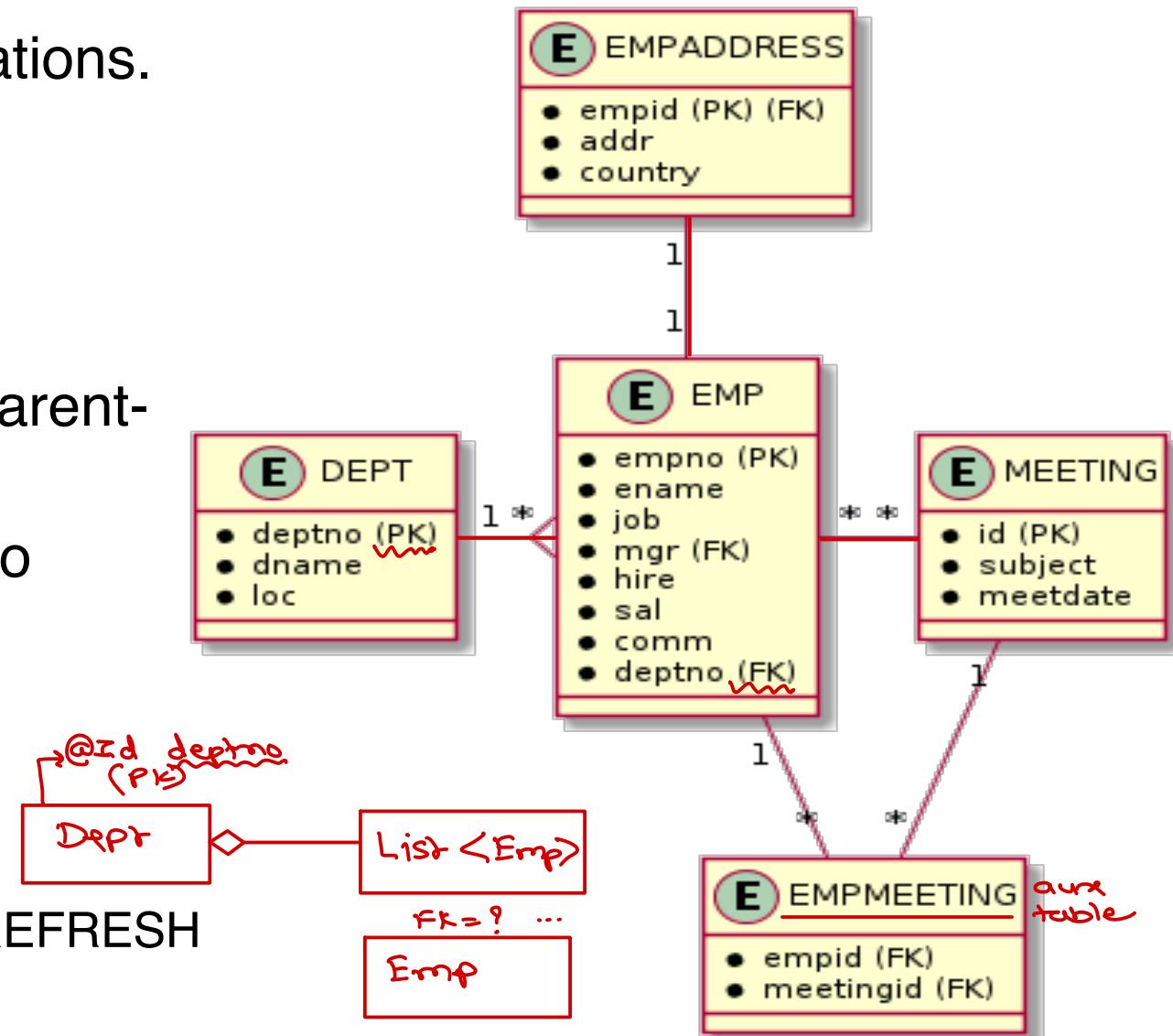
- Hibernate Relations

- Hibernate is ORM tool.
- Java classes/objects are related by means of inheritance and associations.
- Hibernate maps RDBMS table relations to Java class/object relations.
- Hibernate supports both relations
  - Associations ✓
  - Inheritance
- Associations are supported with set of annotations e.g. @OneToMany, @ManyToOne, @ManyToMany, @OneToOne
- Inheritance is supported with set of annotations e.g. @MappedSuperclass, @Inheritance.



# Hibernate Relations (associations)

- Hibernate represents RDBMS table relations.
  - OneToOne
  - OneToMany
  - ManyToOne
  - ManyToMany
- OneToMany & ManyToOne represent parent-child relation between tables.
- Primary key of parent table is mapped to foreign key of child table.
- FetchType
  - Lazy or Eager
- CascadeType
  - PERSIST, MERGE, DETACH, REMOVE, REFRESH



# @OneToMany (uni-directional)

- A Dept has Many Emp.
  - mappedBy – foreign key field in Emp table (that reference to primary key of Dept table).
  - FetchType
    - LAZY: Fetch Dept only (simple SELECT query) and fetch Emp only when empList is accessed (simple SELECT query with WHERE clause on deptno)
    - EAGER: Fetch Dept & Emp data in single query (OUTER JOIN query)
  - CascadeType
    - PERSIST: insert Emp in list while inserting Dept (persist())
    - REMOVE: delete Emp in dept while deleting Dept (remove())
    - DETACH: remove Emp in dept from session while removing Dept from session (detach())
    - REFRESH: re-select Emp in dept while re-selecting Dept (refresh())
    - MERGE: add Emp in dept intion session while adding Dept into session (merge())

```
@Entity @Table(name="dept")
class Dept {
    @Id
    @Column private int deptno; ✓
    @Column private String dname; ✓
    @Column private String loc; ✓
    @OneToMany(mappedBy="deptno")
    private List<Emp> empList;
    // ...
}

@Entity @Table(name="emp")
class Emp {
    @Id
    @Column private int empno; ✓
    @Column private String ename; ✓
    @Column private double sal; ✓
    @Column private int deptno; ✓
    // ...
}
```

The code shows two entity classes: Dept and Emp. The Dept class has a many-to-one relationship to the Emp class, indicated by the `@OneToMany` annotation with `mappedBy="deptno"`. The Emp class has a one-to-many relationship to the Dept class, indicated by the `@ManyToOne` annotation with `mappedBy="deptno"`. Handwritten annotations include red boxes around the `@OneToMany` and `mappedBy` attributes, and a blue circle highlighting the `mappedBy="deptno"` attribute. A blue arrow points from the `mappedBy="deptno"` annotation in the Emp class back to the `deptno` column in the Dept class. A red line with the text "FK field" is drawn from the `deptno` column in the Emp class back to the `deptno` column in the Dept class.



## @ManyToOne (uni-directional)

- Many Emp can have same Dept.
  - FetchType – LAZY or EAGER *\* ← default*
  - CascadeType - PERSIST, MERGE, DETACH, REMOVE, REFRESH
  - @JoinColumn is used along with @ManyToOne to specify foreign key column in EMP table (that reference to primary key of DEPT table).

```
Emp e = session.get(Emp.class, 7900);
System.out.println(e);
System.out.println(e.getDept());
```

```
@Entity @Table(name="dept")
class Dept {
    @Id PK
    @Column private int deptno;
    @Column private String dname;
    @Column private String loc;
    // ...
}
```

~~crossed out~~

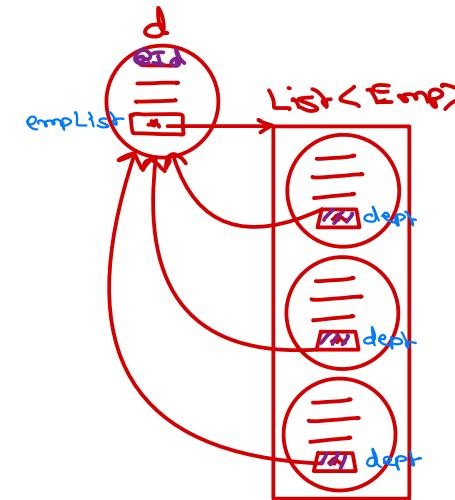
```
@Entity @Table(name="emp")
class Emp {
    @Id
    @Column private int empno;
    @Column private String ename;
    @Column private double sal;
    @Column private int deptno;
    @ManyToOne eager load by default.
    @JoinColumn(name="deptno")
    private Dept dept;
    // ...
}
```

*Fk Column in Emp table*



# @OneToMany and @ManyToOne (bi-directional)

- A Dept have many Emps.
- Many Emp can have same Dept.
- @ManyToOne in Emp class
  - Use @JoinColumn to specify FK column in EMP table.
- @OneToMany in Dept class
  - Use mappedBy to specify FK field in Emp class – now declared as Dept object.
  - FK value is taken from inner Dept object's @Id field.



```
@Entity @Table(name="DEPT")
class Dept {
    @Id
    @Column private int deptno;
    @Column private String dname;
    @Column private String loc;
    @OneToMany(mappedBy="dept")
    private List<Emp> empList;
}
```

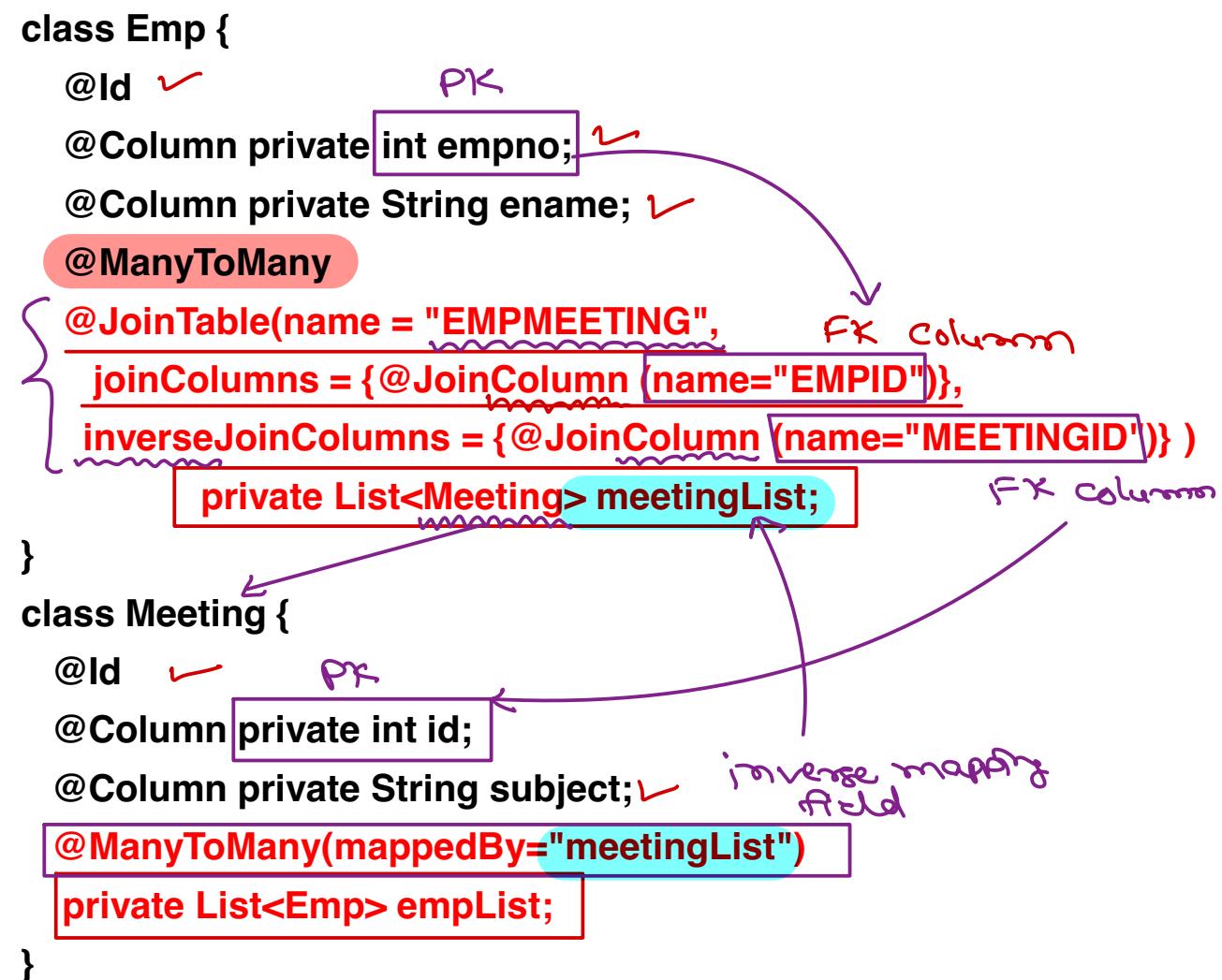
```
@Entity @Table(name="EMP")
class Emp {
    @Id
    @Column private int empno;
    @Column private String ename;
    @Column private double sal;
    @Column private int deptno;
```

```
@ManyToOne
@JoinColumn(name="deptno")
private Dept dept;
```

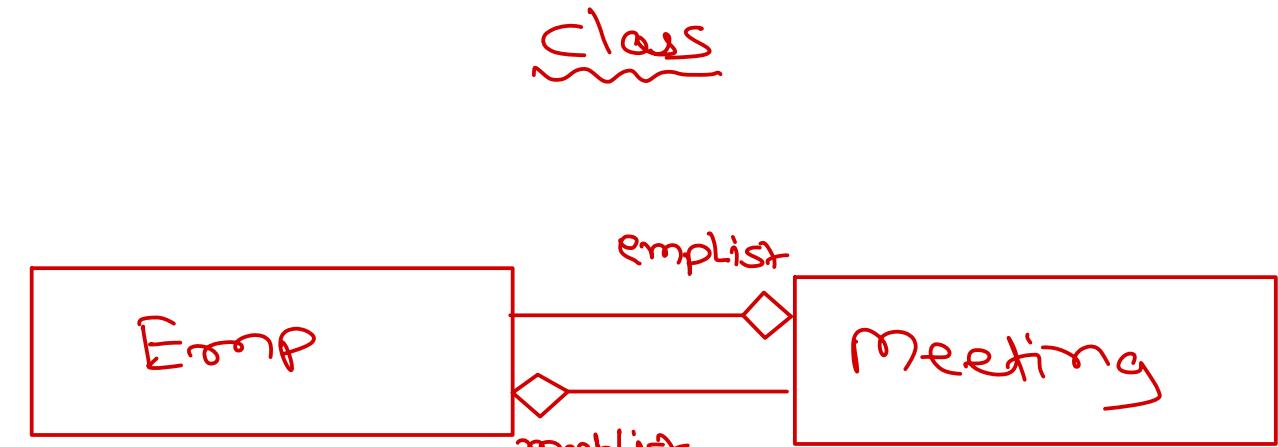
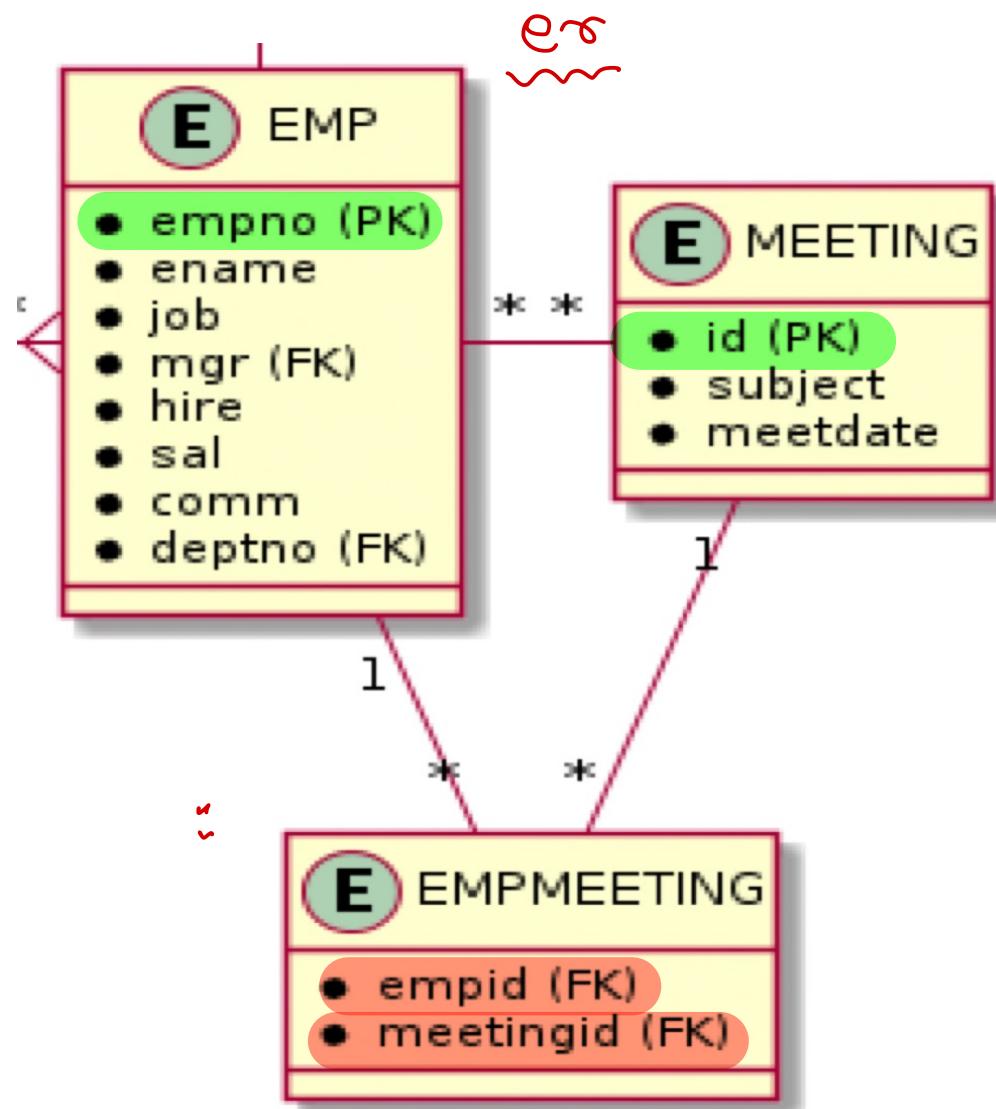
# @ManyToMany (bi-directional)

- One Emp can have many Meetings.
- One Meeting will have many Emps.
- Many-to-many relation is established into two tables via an additional table (auxiliary table).
- The EMP\_MEETING table holds FK of both tables to establish the relation.
- In first class (e.g. Emp) use @ManyToMany along with @JoinTable (refering auxillary table & FK column in it).
  - joinColumn – first table's FK in aux table
  - inverseJoinColumn – second table's FK in aux table
- In second class (e.g. Meeting) use @ManyToMany with mappedBy to setup bi-directional relation.

```
class Emp {  
    @Id ✓  
    @Column private int empno; ✓  
    @Column private String ename; ✓  
    @ManyToMany  
    @JoinTable(name = "EMPMETING",  
    joinColumns = {@JoinColumn (name="EMPID")},  
    inverseJoinColumns = {@JoinColumn (name="MEETINGID")})  
    private List<Meeting> meetingList; ✓  
}  
  
class Meeting {  
    @Id ✓ PK  
    @Column private int id;  
    @Column private String subject; ✓  
    @ManyToMany(mappedBy="meetingList")  
    private List<Emp> empList;  
}
```



The diagram shows the relationship between the Emp and Meeting classes. The Emp class has a many-to-many relationship with Meeting through the EMPMETING join table. The join table has foreign key columns EMPID and MEETINGID. The Meeting class also has a many-to-many relationship with Emp, mapped by the meetingList field. The diagram uses purple annotations to highlight primary keys (PK), foreign keys (FK), and inverse mappings.



## HQL Join

→ also used overriding fetch type given in orm annotations.  
from Emp e join Fetch e.meeting List where —

- SQL joins are used to select data from two related tables.
- Types of joins: Cross, Inner, Left outer, Right outer, Full outer.

### SQL Join syntax:

- SELECT e.ename, d.dname FROM EMP e INNER JOIN DEPT d ON e.deptno = d.deptno;
- ✓ SELECT e.ename, d.dname FROM EMP e, DEPT d WHERE e.deptno = d.deptno;

- HQL joins are little different than SQL joins. They can be used on association so that condition will be generated automatically. Alternatively property of composite object can be given in select or may use traditional join syntax.

### HQL Join syntax:

- SELECT e.ename, d.dname FROM EMP e INNER JOIN e.dept d;  
*entity name*
- SELECT e.ename, e.dept.dname FROM EMP e;  
*entity name*
- SELECT e.ename, d.dname FROM EMP e, DEPT d WHERE e.deptno = d.deptno;

} Object[]

- Note that joins are executed automatically when Hibernate associations are used.



# @OneToOne (bi-directional)

- One Emp have one Address.
- If both tables have same primary key, then use @OneToOne along along with @PrimaryKeyJoinColumn. Use @OneToMany with mappedBy in second class to setup bidirectional relation.
- If a table contains FK for another table use @OneToOne with @JoinColumn.

```
class Emp {  
    @Id  
    @Column private int empno;  
    }  
    {  
        @OneToOne  
        @JoinColumn(name="addr_id")  
        private Address addr;  
    }
```

```
class Address {  
    @Id  
    @Column private int id;  
    @Column private String country;  
    }  
    {  
        @OneToOne(mappedBy = "addr")  
        private Emp emp;  
    }
```

```
class Emp {  
    @Id  
    @Column private int empno;  
    }  
    {  
        @OneToOne  
        @PrimaryKeyJoinColumn  
        private Address addr;  
    }  
  
class Address {  
    @Id  
    @Column private int empid;  
    @Column private String country;  
    }  
    {  
        @OneToOne(mappedBy = "addr")  
        private Emp emp;  
    }
```

# Hibernate Query Language



# Native Queries and HQL

- Ad-hoc SQL queries (on tables) can be executed in hibernate directly in hibernate.
  - `NativeQuery q = session.createSQLQuery(sql);`
- Hibernate recommends using HQL for ad-hoc queries.
- These queries are on hibernate entities (not on tables).
  - `Query q = session.createQuery(hql);`
- HQL supports SELECT, DELETE, UPDATE operation.
- INSERT is limited to `INSERT INTO ... SELECT ...;`



# Hibernate – HQL

- **SELECT**
  - from Book b
  - from Book b where b.subject = :p\_subject
  - from Book b order by b.price desc
  - select distinct b.subject from Book b
  - select b.subject, sum(b.price) from Book b group by b.subject
  - select new Book(b.id, b.name, b.price) from Book b
- **DELETE**
  - delete from Book b where b.subject = :p\_subject
- **UPDATE**
  - update Book b set b.price = b.price + 50 WHERE b.subject = :p\_subject
- **INSERT**
  - insert into Book(id, name, price) select id, name, price from old\_books



# Named queries

- Using multiple HQL queries in project will scatter them into multiple files and thus application maintenance become complicated.
- All queries related to an entity can be associated with the class using `@NamedNativeQuery` or `@NamedQuery`.
- `NamedNativeQuery` represent SQL query.
  - Use `session.getNamedNativeQuery()` to access `NativeQuery`.
  - Invoke methods on `NativeQuery` to perform appropriate operations.
- `NamedQuery` represent HQL query.
  - Use `session.getNamedQuery()` to access `NativeQuery`.
  - Invoke methods on `NativeQuery` to perform appropriate operations.
- To associate multiple queries use `@NamedNativeQueries` or `@NamedQueries`.



# HQL Join

- SQL joins are used to select data from two related tables.
- Types of joins: Cross, Inner, Left outer, Right outer, Full outer.
- SQL Join syntax:
  - `SELECT e.ename, d.dname FROM EMP e INNER JOIN DEPT d ON e.deptno = d.deptno;`
  - `SELECT e.ename, d.dname FROM EMP e, DEPT d WHERE e.deptno = d.deptno;`
- HQL joins are little different than SQL joins. They can be used on association so that condition will be generated automatically. Alternatively property of composite object can be given in select or may use traditional join syntax.
- HQL Join syntax:
  - `SELECT e.ename, d.dname FROM EMP e INNER JOIN e.dept d;`
  - `SELECT e.ename, e.dept.dname FROM EMP e;`
  - `SELECT e.ename, d.dname FROM EMP e, DEPT d WHERE e.deptno = d.deptno;`
- Note that joins are executed automatically when Hibernate associations are used.



# Hibernate Stored Procedures



# Hibernate 3 – Calling Stored Procedure

- Use `@NamedNativeQuery` to define stored procedure call statement.
  - `@NamedNativeQuery(name="qry", query="CALL books_in_range(:p1,:p2)", hints = {@QueryHint ( name = "org.hibernate.callable", value = "true" ) })`
- Simple SP (with no out params) can be executed like named queries.
- If SP returns rows (due to SELECT), `resultClass` can be set in `@NamedNativeQuery`.



# Hibernate 3 – Calling Stored Procedure

---

- Hibernate3 doesn't have support for out params, so JDBC code can be embedded using doWork() or doReturningWork() method.
- They respectively use Work (returns void) and ReturningWork<?> (returns value) functional interfaces having jdbc Connection as arg.



# Hibernate 5 – Calling Stored Procedure

- Hibernate5 provides ProcedureCall object to deal with SP.
  - Way1:  
`session.createStoredProcedureCall()`
  - Way2: use  
`@NamedStoredProcedureQuery` on entity class and  
`session.getNamedProcedureCall()` similar to named queries.
- Procedure call deal with out params using  
`registerStoredProcedureParameter()` and setting param mode as IN or OUT.
- This is JPA compliant.



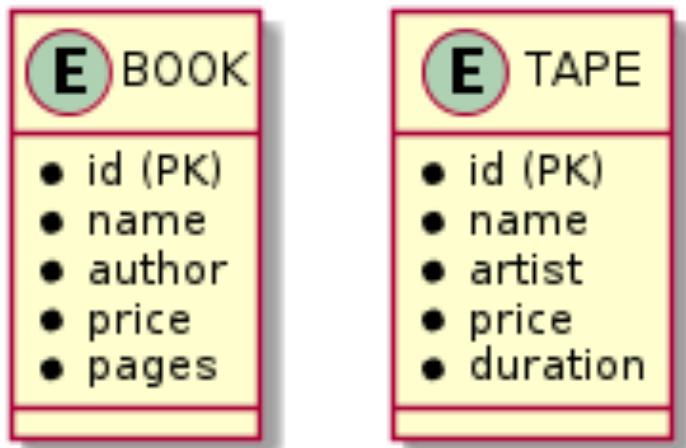
# Hibernate Entity Inheritance



# Hibernate Relations (inheritance)

- **@MappedSuperclass**

- Different tables for sub-class entities.
- Easy to implement.
- Super class is not entity.
- No polymorphic queries.
- Relations/mapping in super class are applicable only to sub-class table (as super-class doesn't have table).



**@MappedSuperclass**

```
class Product {  
    @Id  
    @Column private int id;  
    @Column private String name;  
    @Column private double price;  
    // ...  
}
```

**@Entity**

```
@Table(name="BOOK")  
class Book extends Product {  
    @Column private String author;  
    @Column private int pages;  
    // ...  
}
```

**@Entity**

```
@Table(name="TAPE")  
class Tape extends Product {  
    @Column private String artist;  
    @Column private int duration;  
    // ...  
}
```

# Hibernate Inheritance – SINGLE\_TABLE strategy

- **SINGLE\_TABLE**

- Single table for all sub-class entities.
- Easy to implement.
- All classes including super class are entity classes.
- Polymorphic queries, High performance.
- Relations in super class..



```
@Entity
@Table(name="PRODUCT")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE")
class Product {
    @Id
    @Column private int id;
    @Column private String name;
    @Column private double price;
    // ...
}

@Entity
@DiscriminatorValue("BOOK")
class Book extends Product {
    // ...
    @Column(name="author")
    private String author;
    @Column(name="info")
    private int pages;
}

@Entity
@DiscriminatorValue("TAPE")
class Tape extends Product {
    // ...
    @Column(name="author")
    private String artist;
    @Column(name="info")
    private int duration;
}
```



# Hibernate Inheritance

- TABLE\_PER\_CLASS
  - Different table for each sub-class & separate table for super-class. Super-class table is very small.
  - All classes including super class are entity classes.
  - Joins are fired to fetch data of sub-class entities.
  - Polymorphic queries, less performance, complex join queries.
  - Relations in super class.
- JOINED
  - Different table for each sub-class & separate table for super-class. Super-class table contains common data. Sub-class tables contains only specific data.
  - All classes including super class are entity classes.
  - Joins are fired to fetch data of sub-class entities.
  - Polymorphic queries, Join queries.
  - Relations in super class.



# Hibernate Fwd & Rev Engg



# Forward and Reverse Engineering

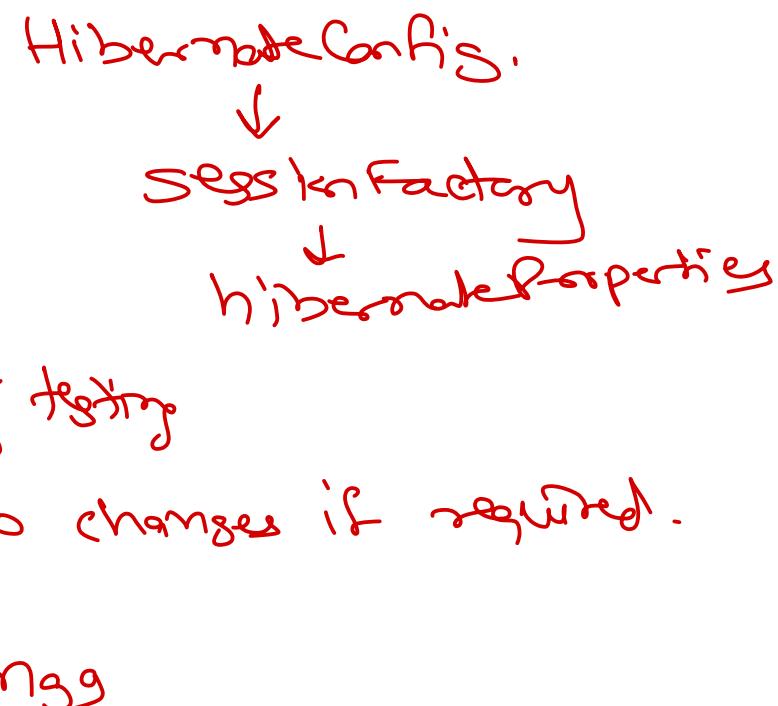
- There are two approaches
  - Database first approach (reverse engg)
  - Entity first approach (forward engg)
- Reverse engg
  - 1. Design database and create tables with appropriate relations. (PK + FK)
  - 2. Implement entity classes with appropriate ORM annotations
- Two ways to implement reverse engg
  - Manual approach
  - Wizard/tools (Eclipse -> JPA project)

JBoss → Hibernate Tools  
Plugin



# Forward and Reverse Engineering

- Forward engg
  - 1. Implement entity classes with appropriate ORM annotations.
  - 2. Database tables (with corresponding relations) will be automatically created when program is executed.
- Hibernate.cfg.xml – hibernate.hbm2ddl.auto
  - create: Db dropping will be generated followed by database creation.
  - create-only: Db creation will be generated.
  - create-drop: Drop the schema and recreate it on SessionFactory startup. Additionally, drop the schema on SessionFactory shutdown.
  - update: Update the database schema. → do changes if required.
  - validate: Validate the database schema
  - none: No action will be performed. → dev engg



# Hibernate Caching



# Hibernate caching

- Hibernate caches are used to speed up execution of the program by storing data (objects) in memory and hence save time to fetch it from database repeatedly.
- There are two caches
  - Session cache (L1 cache) ✓
  - SessionFactory cache (L2 cache)



# Hibernate session cache

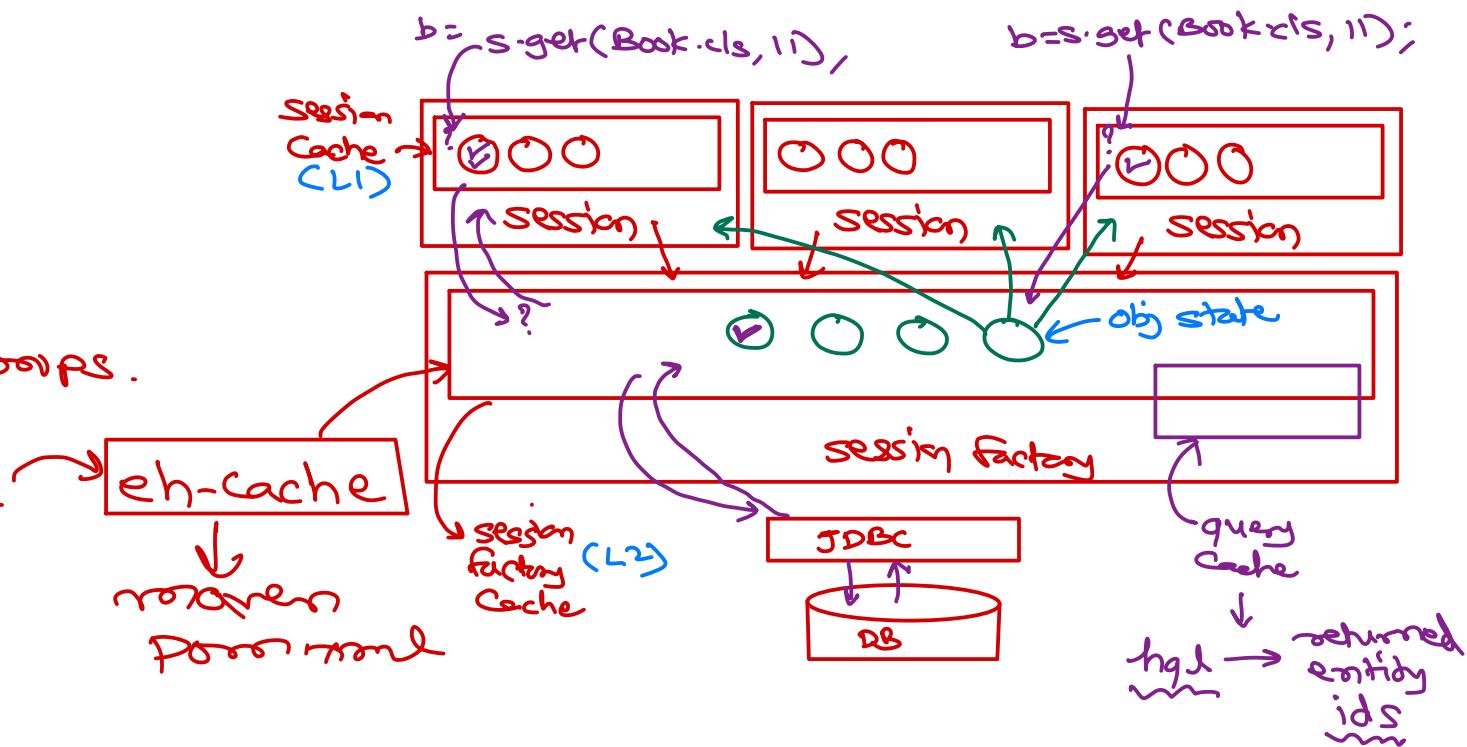
- Collection of entities per session – persistent objects.
- Hibernate keep track of state of entity objects and update into database.
- Session cache cannot be disabled. *Part of hibernate*
- If object is present in session cache, it is not searched into session factory cache or database.
- Use refresh() to re-select data from the database forcibly.



# Hibernate session factory cache

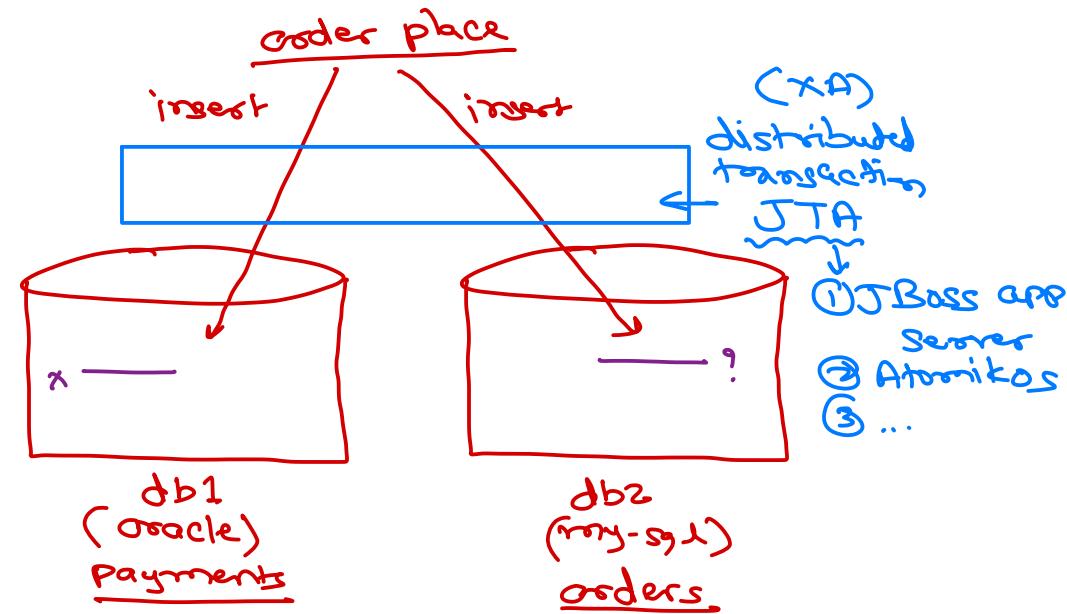
- Collection of entities per session factory (usually single in appln).
- By default disabled, but can be enabled and configured into `hibernate.cfg.xml` or `hibernate poors`.
- Need to add respective additional second cache jars in project and optional cache config file (e.g. `ehcache.xml`). *how many objs. expiration policy.*

```
<property name="hibernate.cache.region.factory_class">  
    org.hibernate.cache.ehcache.EhCacheRegionFactory  
</property>  
<property name="hibernate.cache.use_second_level_cache"> true </property>
```



# Hibernate session factory cache

- Four types of second level caches:  
EHCache, Swarm Cache, OS Cache,  
Tree Cache
- Use @Cache on entity class to cache  
its objects.
- Decide cache policy for those object  
and specify into its usage attribute:  
READ\_ONLY, READ\_WRITE,  
NONSTRICT\_READWRITE,  
TRANSACTIONAL
- Stores the objects into the map (with  
its id as key). So lookup by key is very  
fast.



# Hibernate session factory cache

- Cache Strategies (enum CacheConcurrencyStrategy)
  - READ\_ONLY: Used only for entities that never change (exception is thrown if an attempt to update such an entity is made). It is very simple and performant. Very suitable for some static reference data that don't change.
  - NONSTRICT\_READ\_WRITE: Cache is updated after a transaction that changed the affected data has been committed. Thus, strong consistency is not guaranteed and there is a small time window in which stale data may be obtained from cache. This kind of strategy is suitable for use cases that can tolerate eventual consistency.
  - READ\_WRITE: This strategy guarantees strong consistency which it achieves by using 'soft' locks: When a cached entity is updated, a soft lock is stored in the cache for that entity as well, which is released after the transaction is committed. All concurrent transactions that access soft-locked entries will fetch the corresponding data directly from database.
  - TRANSACTIONAL: Cache changes are done in distributed XA transactions. A change in a cached entity is either committed or rolled back in both database and cache in the same XA transaction.



# Hibernate session factory cache

- Cache Types
  - EHCache: It can cache in memory or on disk and clustered caching and it supports the optional Hibernate query result cache.
  - Swarm Cache: A cluster cache based on JGroups. It uses clustered invalidation, but doesn't support the Hibernate query cache.
  - OSCache (Open Symphony Cache): Supports caching to memory and disk in a single JVM with a rich set of expiration policies and query cache support.
  - JBoss Tree Cache: A fully transactional replicated clustered cache also based on the JGroups multicast library. It supports replication or invalidation, synchronous or asynchronous communication, and optimistic and pessimistic locking.
  - Note that not all cache support all strategies.



# Hibernate session factory cache

Cache	Read Only	NonStrict Read-Write	Read-Write	Transactional
EH Cache	Yes	Yes	Yes	No
OS Cache	Yes	Yes	Yes	No
Swarm Cache	Yes	Yes	No	No
Jboss Cache	Yes	No	No	Yes ✓



# Hibernate Advanced problems



# OpenSessionInView

- Usually Tx is closed in dao/bean layer.
- If entity contains some lazily loaded collections and they are accessed from view (JSP page), then application fails with LazyInitializationException; because no active tx is present in view.
- To handle this session can be opened from view itself. This is done using
  - ```
<property  
    name="hibernate.enable_lazy_load_no_trans">  
    true </property>
```
- This is called as "OpenSessionInView" pattern.
- But now no clear separation in dao layer and view layer. View directly request to database. If it fails, exception cannot be handled. This is not recommended.



# N+1 Problem

- While fetching n records from a table, due to ManyToOne or OneToOne relation (eager fetch) parent entity will be fetched n times. This is referred as N+1 problem.
  - select e from Emp e;
- This problem can be handled by specifying fetch mode in the query.
  - select e from Emp e join fetch e.dept;
- N+1 problem is also observed when session factory cache is disabled, but query cache is enabled. In this case query cache save only ids and for each id data is re-fetched from database.



# JPA

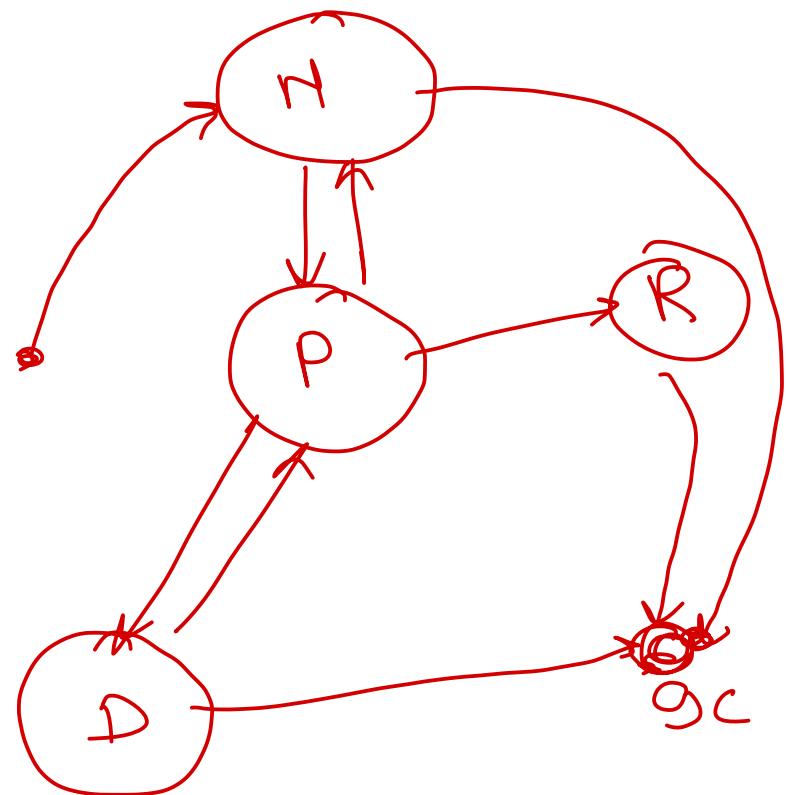


Sunbeam Infotech

[www.sunbeaminfo.com](http://www.sunbeaminfo.com)

# JPA

- JPA is specification for ORM.
- All ORM implementations follow JPA specification.
  - Hibernate, Torque, iBatis, EclipseLink, ...
- Hibernate implements JPA specs.
  - SessionFactory → EntityManagerFactory
  - Session → EntityManager
    - find(), persist(), merge(), refresh(), remove(), detach(), ...
  - HQL → JPQL
  - hibernate.cfg.xml → persistence.xml
- JPA versions (Standard)
  - 1.0, 1.1, 2.0, 2.1, 2.2
- JPA Entity life cycle
  - New (Transient)
  - Managed (Persistent)
  - Detached (Detached)
  - Removed (Removed)



- Hibernate Criteria
  - Only for SELECT operation
  - Deprecated
- Hibernate Query
  - HQL query
  - Parsed & converted to SQL
- CriteriaQuery
  - JPA complaint
  - Use builder design pattern
  - For SELECT operation
- CriteriaUpdate for update operation
- CriteriaDelete for delete operation
- CriteriaQuery

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Book> crQuery =
builder.createQuery(Book.class);
Root<Book> table = crQuery.from(Book.class);
crQuery.select(table)
.where(builder.equal(table.get("author"), author));
TypedQuery<Book> q = em.createQuery(crQuery);
list = q.getResultList();
```

# Spring JPA



# Spring JPA Integration

- Spring DI simplifies JPA.
- LocalEntityManagerFactoryBean bean provides entity manager factory, while transaction automation is done by JpaTransactionManager bean.
- Steps:
  - In pom.xml, add spring-orm, mysql-connector-java, hibernate-core.
  - Configure META-INF/persistence.xml.
  - Create entityManagerFactory (with JPA PersistenceUnitName configured), transactionManager beans. Also set default transactionManager.
  - Implement entity classes.
  - Implement @Repository class & auto-wire entity manager using @PersistentContext EntityManager em & then perform operations.
  - Implement @Service layer and mark business logic methods as @Transactional.
    - Note that single business operation (from service layer) may deal with multiple operations on different repositories. @Transactional put all ops under same tx.



# Spring JPA Integration



Sunbeam Infotech

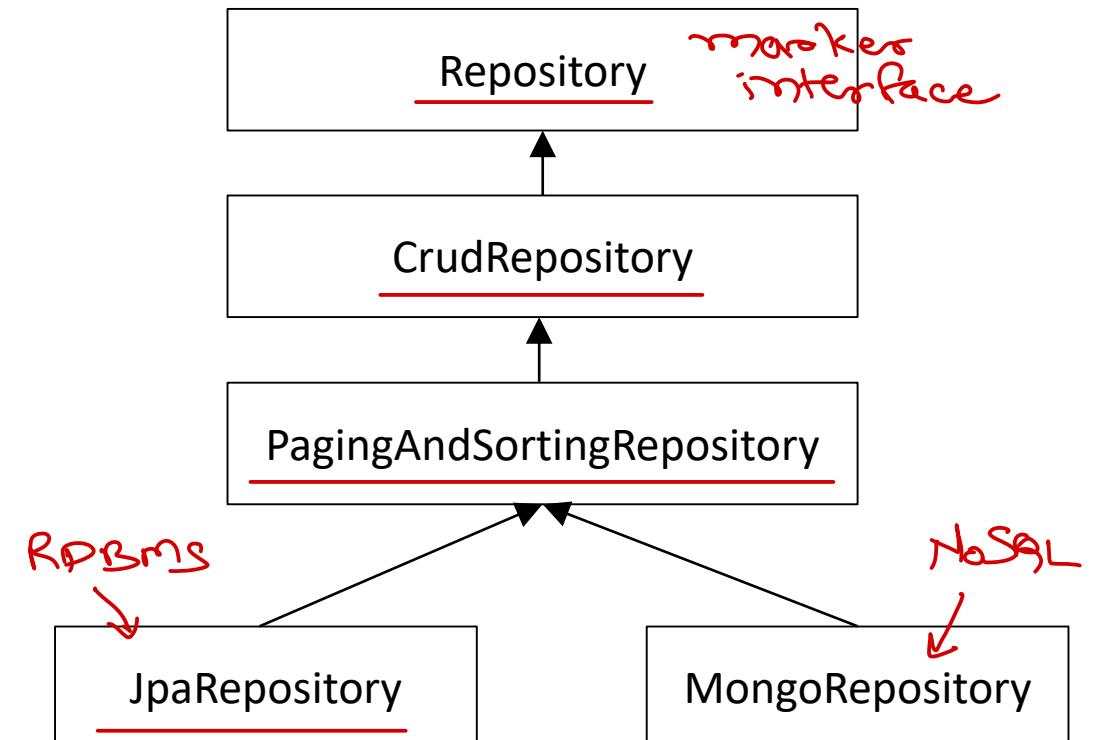
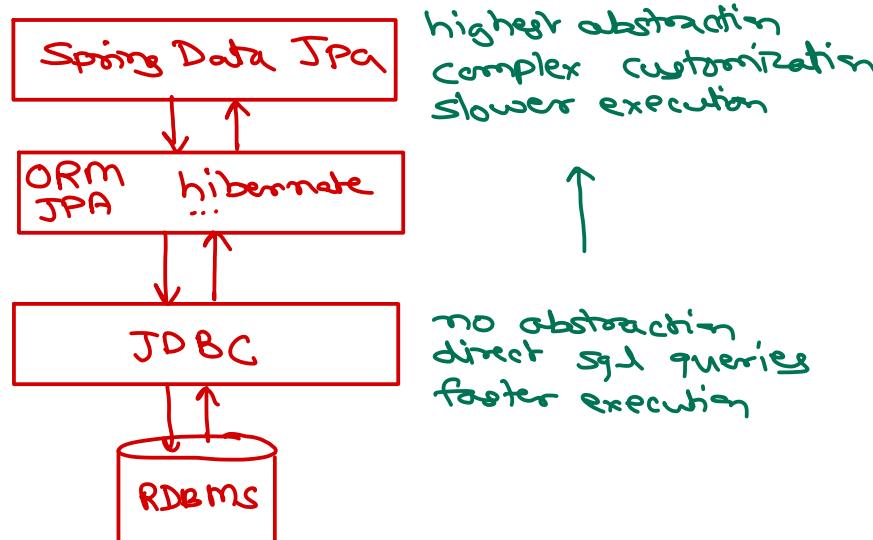
[www.sunbeaminfo.com](http://www.sunbeaminfo.com)

# Spring Data

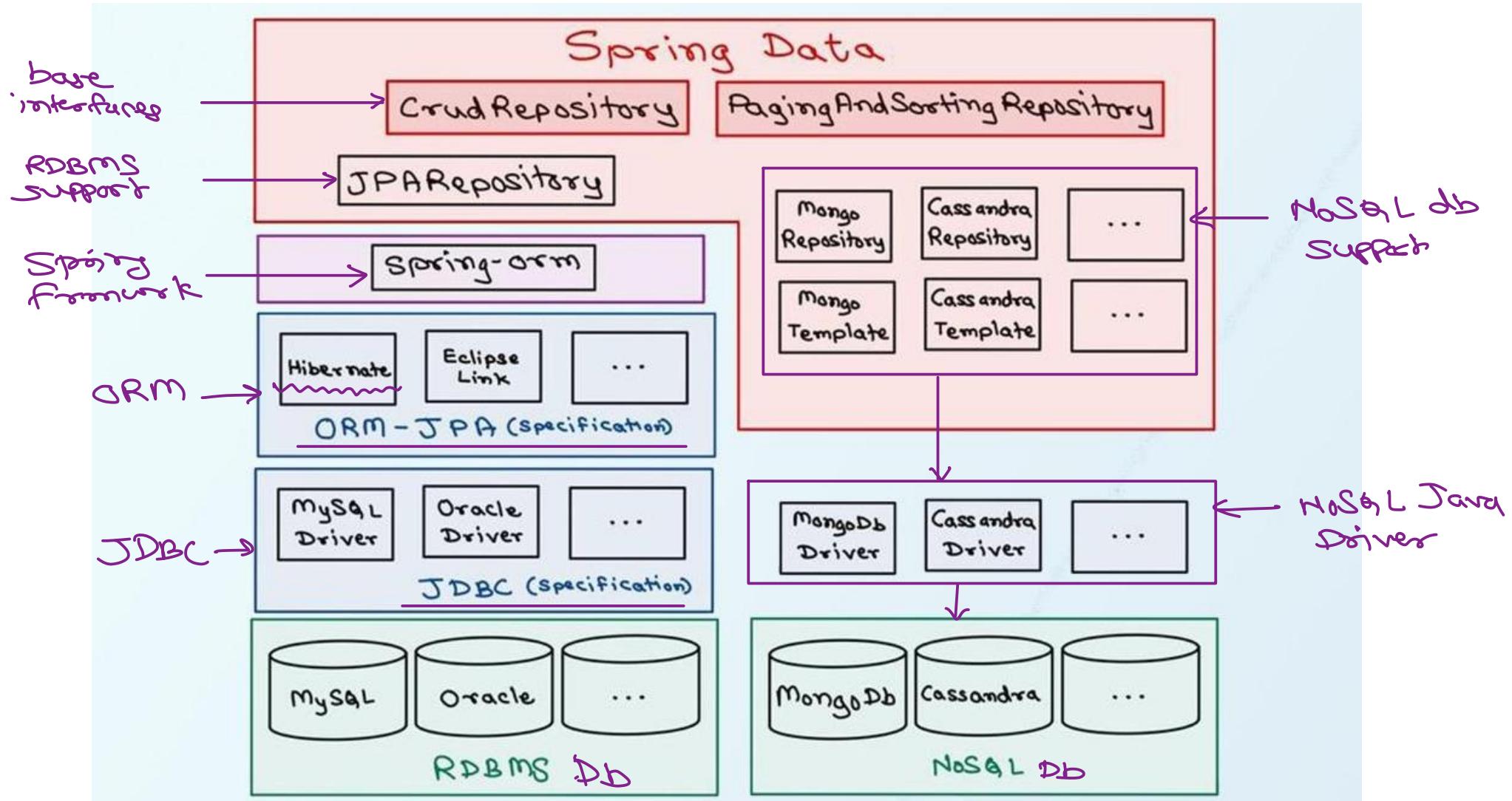


# Spring Data

- Spring Data provides unified access to databases -- RDBMS or NoSQL.
- Spring Data JPA provides repository support for the Java Persistence API (JPA) -- RDBMS.
- Advantages
  - Eases development of applications that need to access JPA data sources.
  - Lot of boilerplate/repeated code is eliminated.
  - Consistent configuration and unified data access.



# Spring Data



# Spring Data

Cloud Repository <>

- Available methods provide basic CRUD operations.
- For application specific database queries build queries using keywords.
  - Query expressions are usually property traversals combined with concatenation operators And / Or as well as comparison operators Between, LessThan, Like, etc.
  - IgnoreCase for individual properties or all properties.
  - OrderBy for static ordering.
- Query method examples
  - Customer Customer findByname(String name);
  - List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
  - List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  - List<Person> findByLastnameIgnoreCase(String lastname);
  - List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  - Stream<Person> findByAgeBetween(int minAge, int maxAge);

By  
And  
where

Properties in  
entity class



# Spring Data

- Query method examples

- User findFirstByOrderByLastnameAsc();
- User findTopByOrderByAgeDesc();
- List<Person> findByAddressZipCode(ZipCode zipCode);
- Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);
- List<User> findFirst10ByLastname(String lastname, Sort sort);
- List<User> findTop10ByLastname(String lastname, Pageable pageable);

- Custom query methods

- Custom queries can be added using @Query annotation.

- JPQL (HQL) or SQL queries can be used

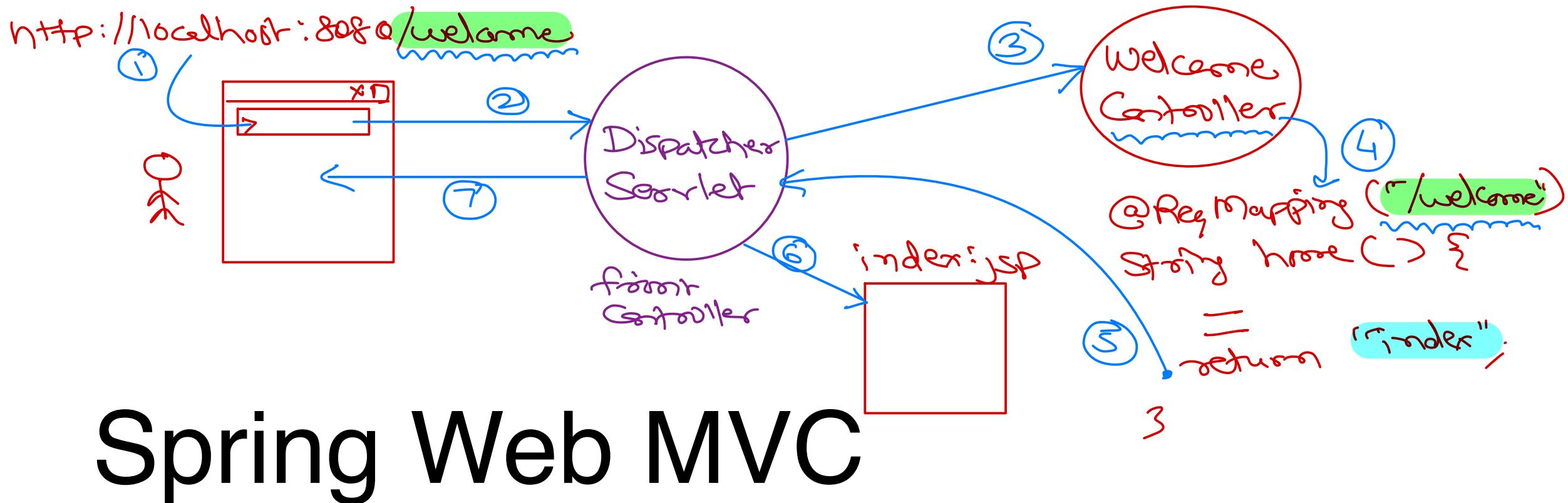
- `@Query("select distinct b.subject from Book b")`

```
List<String> findDistinctSubjects();
```

- `@Query(value="select distinct b.subject from BOOKS b", native=true)`

```
List<String> findDistinctSubjectsWithSQL();
```





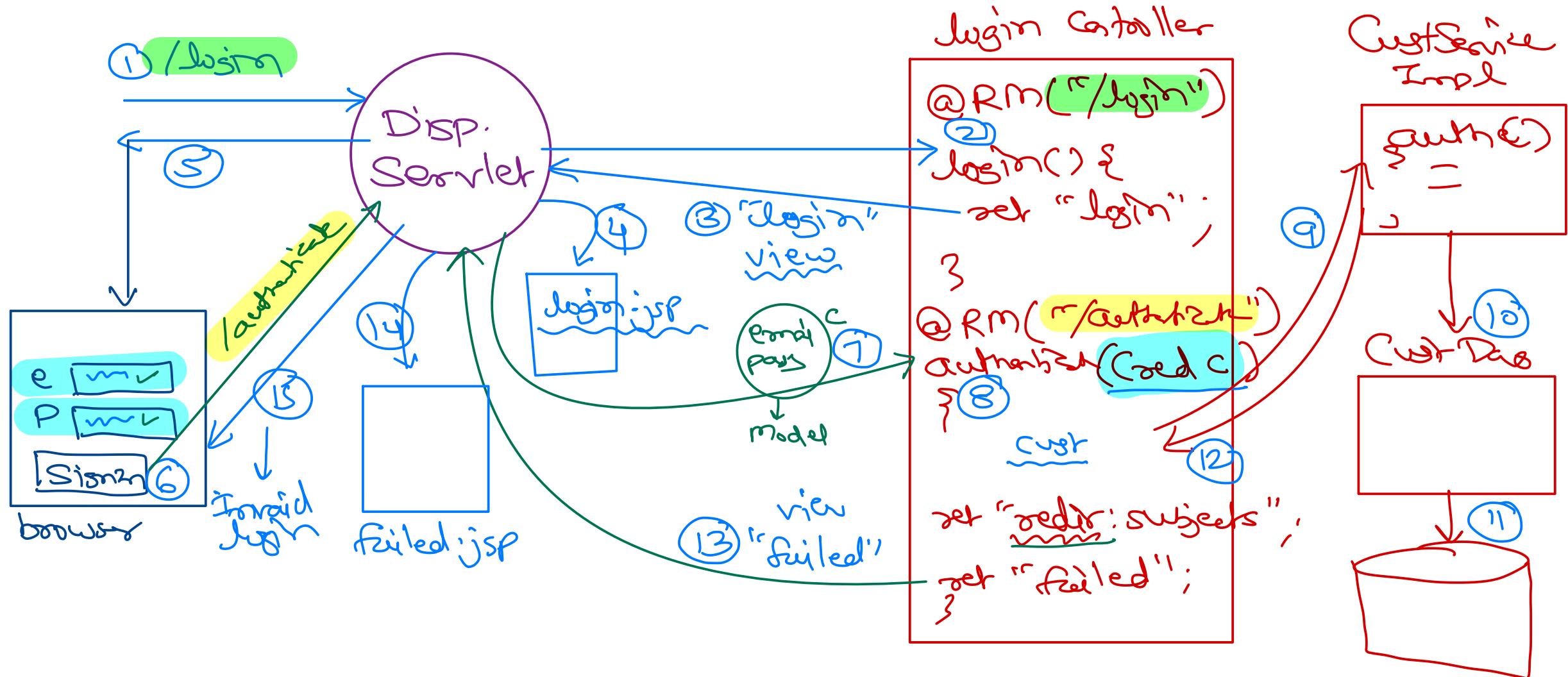
# Spring Web MVC

# Spring Web MVC

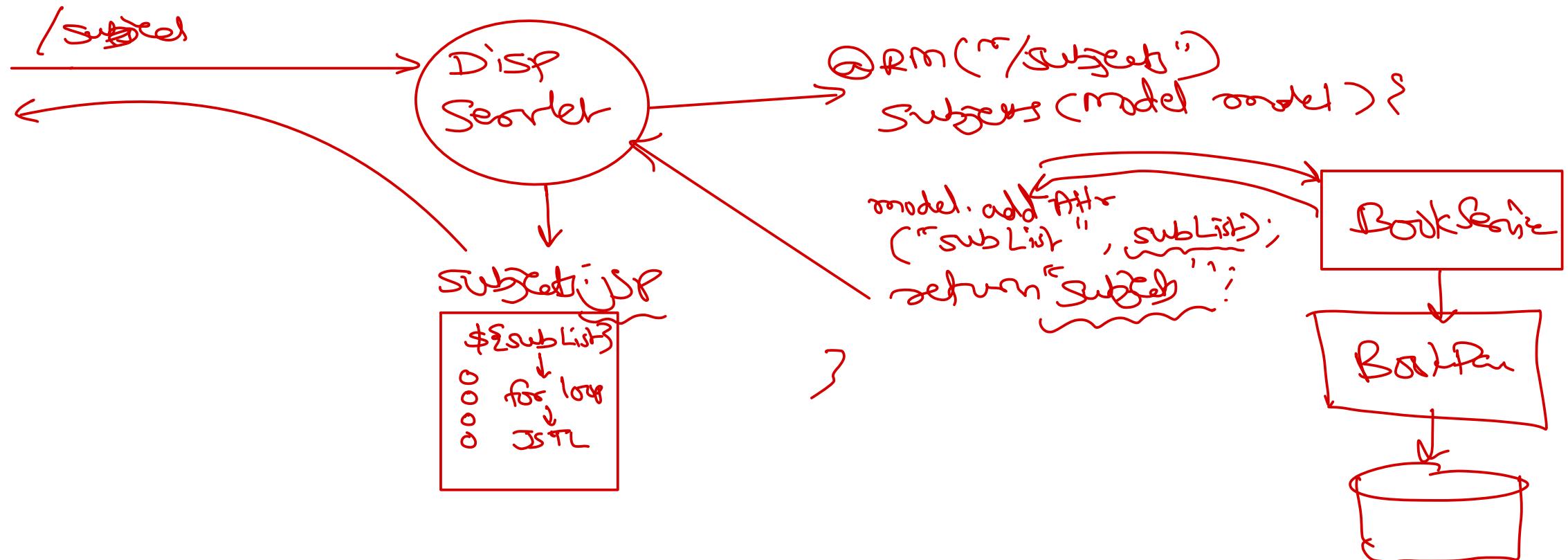
- MVC is design-pattern.
  - Divide application code into multiple relevant components to make application maintainable and extendable.
  - M: Model: Data of the application.
  - V: View: Appearance of data.
  - C: Controller: Interaction between models & views.
- Typical MVC implementation using Servlets & JSP.
  - Model: Java beans
  - View: JSP pages
  - Controller: Servlet dispatching requests
- Spring MVC components
  - Model: POJO classes holding data between view & controller.
  - View: JSP pages
  - Controller: Spring Front Controller i.e. DispatcherServlet
  - User defined controller: Interact with front controller to collect/send data to appropriate view, process with service layer.



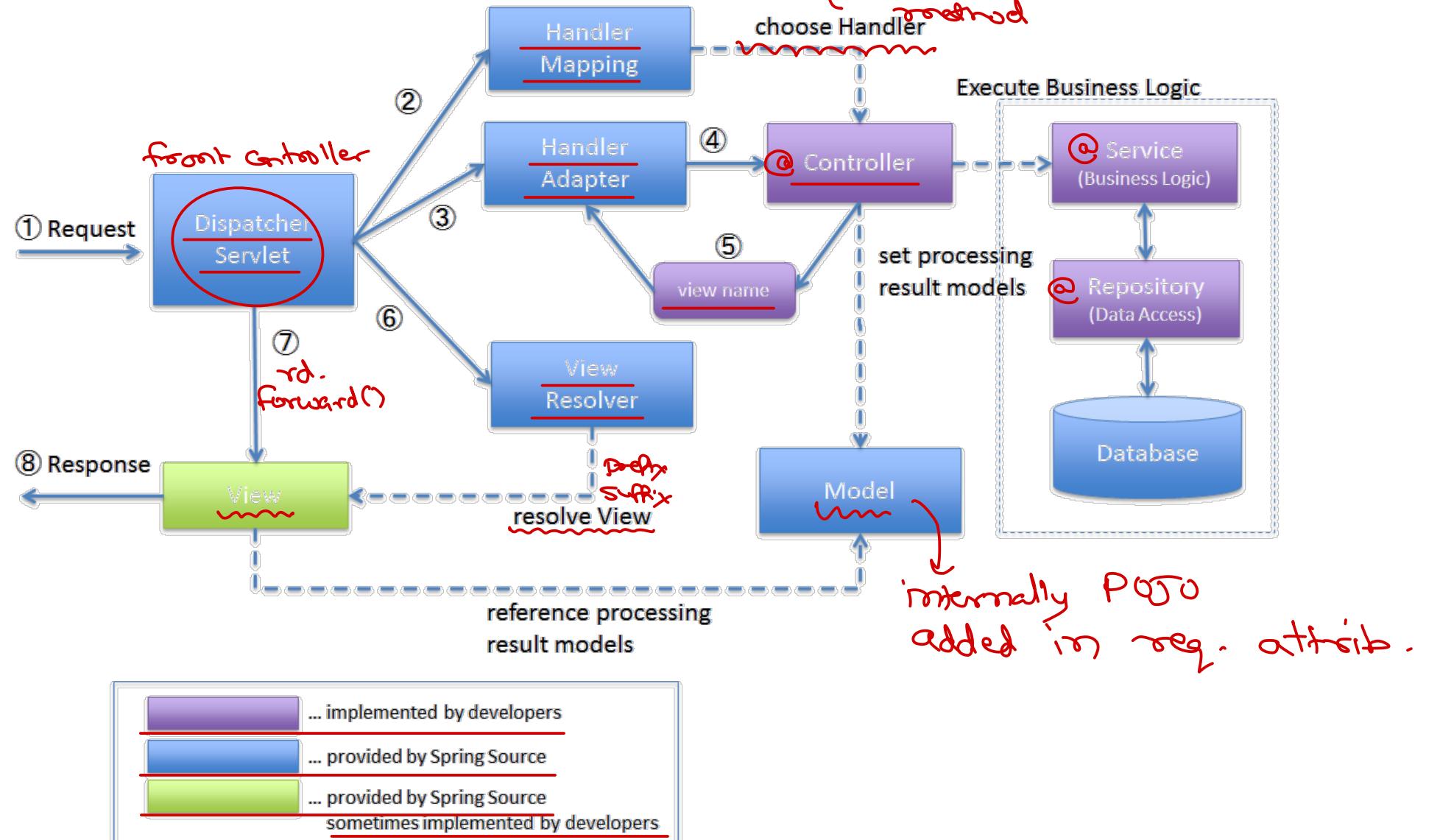
# Spring Web MVC



# Spring Web MVC



# Spring Web MVC



# Spring Web MVC

- DispatcherServlet receives the request.
  - DispatcherServlet dispatches the task of selecting an appropriate controller to HandlerMapping. HandlerMapping selects the controller which is mapped to the incoming request URL and returns the (selected Handler) and Controller to DispatcherServlet.
  - DispatcherServlet dispatches the task of executing of business logic of Controller to HandlerAdapter.
  - HandlerAdapter calls the business logic process of Controller.
  - Controller executes the business logic, sets the processing result in Model and returns the logical name of view to HandlerAdapter.
  - DispatcherServlet dispatches the task of resolving the View corresponding to the View name to ViewResolver. ViewResolver returns the View mapped to View name.
  - DispatcherServlet dispatches the rendering process to returned View.
  - View renders Model data and returns the response.
- 
- <https://terasolunaorg.github.io/guideline/1.0.x/en/Overview/SpringMVCOversview.html>



# Spring Web MVC Annotation Config (not using Spring boot) → maven pom

war → maven

- pom.xml: properties → <failOnMissingWebXml>false</failOnMissingWebXml>
- web.xml is replaced by MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer ← given by Spring mvc
  - Encapsulate declaration of dispatcher servlet.
  - Required for creating spring container/context.
    - getRootConfigClasses(): return config class for root webapplicationcontext.
    - getServletConfigClasses(): return config class for servlet webapplicationcontext
    - getServletMappings(): return dispatcher servlet url pattern (/)
- spring5-servlet.xml is replaced by MyWebMvcConfig implements WebMvcConfigurer
  - @Configuration
  - @ComponentScan(...): scans stereo-type annotations + other @Configuration classes.
  - @EnableWebMvc: creates AnnotationConfigWebApplicationContext.
  - @Bean: viewResolver → UrlBasedViewResolver bean  
- prefix , suffix

@Controller  
views



# Spring Boot Web MVC

- Create new Spring Boot project with Spring Web starter.
- In pom.xml add JSP & JSTL support
  - org.apache.tomcat.embed - tomcat-embed-jasper: provided
  - javax.servlet - jstl
- Configure viewResolver in application.properties
  - spring.mvc.view.prefix=/WEB-INF/jsp/
  - spring.mvc.view.suffix=.jsp
- Implement a controller with a request handler method returning view name (input).
- Under src/main create directory structure webapp/WEB-INF/jsp.
- Create input.jsp under webapp/WEB-INF/jsp to input user name and submit to server.
- Add another request handler method in controller to accept request param and send result to output page.
- Create output.jsp under webapp/WEB-INF/jsp to display the result.



# Request handler method → Inside @Controller class

- @RequestMapping attributes
  - value/path = “url-pattern”
  - method = GET | POST | PUT | DELETE
  - params/header = ... (map request only if given param or header is present).
  - consumes = ... (map request only if given request body type is available).
  - produces = ... (produce given response type from handler method)
- One request handler method can be mapped to multiple request methods.
- One request handler method can be restrict to set of request methods.
- To restrict handler method to single request method, shorthand annotations available.
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping



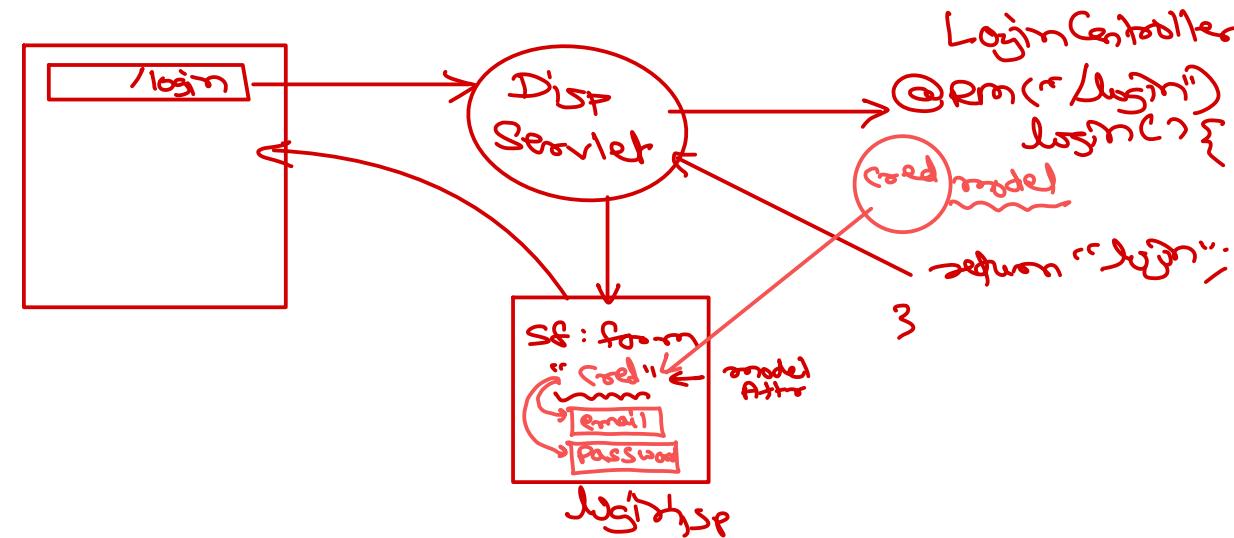
# Request handler method

- An @RequestMapping handler method can have a very flexible signatures.
- Supported method argument types
  - HttpServletRequest, HttpServletResponse
  - @RequestParam, @RequestHeader, @PathVariable
  - Map or Model or ModelMap → to send data to view.
  - Command object, @ModelAttribute, Errors or BindingResult
  - InputStream, OutputStream
  - HttpSession, @CookieValue, Locale
  - HttpEntity<>, @RequestBody.
- Supported method return types
  - String, View, Model or Map, ModelAndView
  - HttpHeaders, void
  - HttpEntity<>, ResponseEntity<>, @ResponseBody.



# Using spring tags

- Spring can work well with JSP and JSTL tags.
- Spring also have its own set of custom tags mainly for HTML input form, localization and validation.
- `<%@ taglib prefix="sf"  
uri="http://www.springframework.org/tags/form" %>`
- `<sf:form modelAttribute="command" action="auth">` ✓
  - Form backing bean or command object.
    - `<sf:input path="email"/>`
    - `<sf:password path="password"/>` ✓
- The *command* is model (POJO) object that carry data between view and controller.
- Model objects are request scoped.





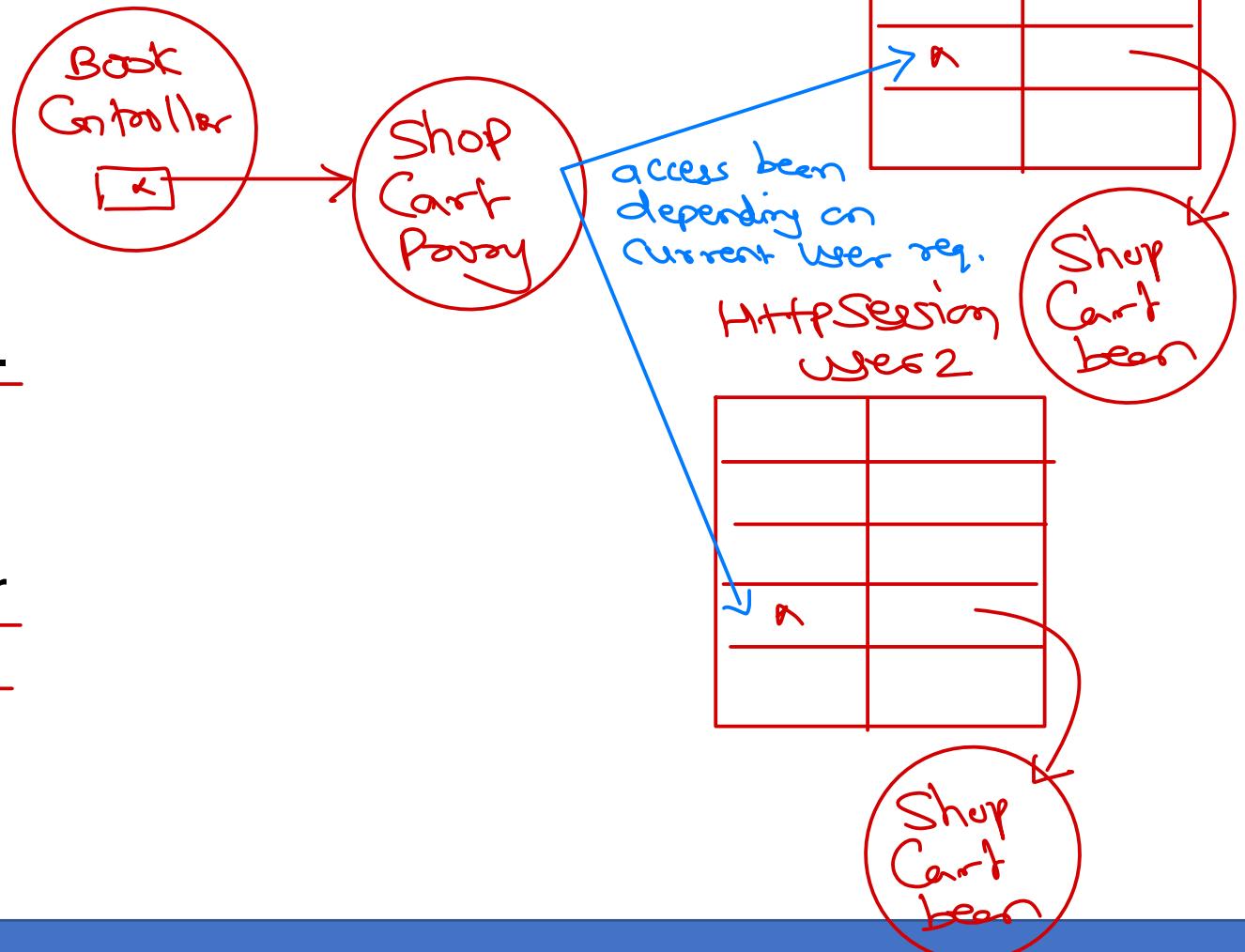
Sunbeam Infotech

[www.sunbeaminfo.com](http://www.sunbeaminfo.com)

# Session and Request scoped beans

HttpSession  
User1

- Spring bean scopes
  - Singleton
  - Prototype
  - Session
  - Request
- Session and Request scope beans are only possible in web applications.
- The scope management is done via proxies. (internally)
- Bean proxy is auto-wired in controller class. Depending on session/request internally new bean is created and made available in that context.



# Spring Static resources

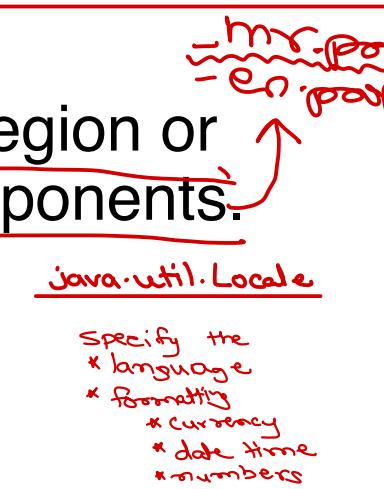
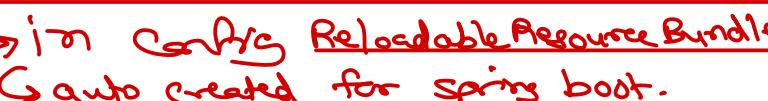
- Static resources like JS, CSS, images should be mapped to some location.
- <mvc:resources location="/WEB-INF/static/css/" mapping="/css/\*\*"/>



Spring Boot auto configures path of static resources  
to resources/static folder.

# Internationalization / Localization

- Adapting computer software to different languages, regional peculiarities and technical requirements of a target locale.
- Internationalization is the process of designing a software application so that it can be adapted to various languages and regions. → `mess.properties`  

- Localization is the process of adapting internationalized software for a region or language. Localization is done for each locale by added respective components.  

- Current locale can be accessed in request handler as Locale argument.
- Spring application steps  

  - Create messageSource bean and provide base path of properties file.
  - Create properties file for different locale.  

  - In views add <s:message code="property-key-name"/>. 
- Current locale can be stored using SessionLocaleResolver or CookieLocaleResolver.
- Locale can be changed dynamically using LocaleChangeInterceptor (configured using <mvc:interceptors/>).

# Spring MVC validation

- For web applications client side validations are preferred for better user experience
- However client side validations can be bypassed/skipped by client.
- To ensure only validity of data provide server side validations (as well).
- Spring validation framework helps for server side validations.
- Spring supports JSR-303 validations.
  - @NotBlank, @NotEmpty, @Size, @Email, @Pattern, @DateTimeFormat, @Min, @Max
- Steps:
  - Add dependency hibernate-validator in pom.xml *(In spring boot → validator starters)*
  - Use appropriate annotations on model classes.
  - Use @Valid on @ModelAttribute in request handler method.
  - Next argument in request handler should be BindingResult method.
  - Use <sf:errors/> in view to show error codes.



# Spring exception handler

- Implement user defined exception class.
- In controller request handler method use `@ExceptionHandler({MyException.class})`.
- Configure exception mapping in spring config.

```
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <prop key="java.lang.RuntimeException">error</prop>
        </props>
    </property>
    <property name="defaultErrorView" value="error" />
</bean>
```

this bean is  
auto configured  
in Spring bean.



## @ControllerAdvice

→ global exception handler  
for multiple controllers.

use AOP internally.





Sunbeam Infotech

[www.sunbeaminfo.com](http://www.sunbeaminfo.com)

# File upload and download

- File upload
  - Add commons-fileupload dependency in pom.xml.
  - In spring mvc config add CommonsMultipartResolver bean.
  - In view form, add HTML file input tag. Make form enctype="multipart/form-data", method="post" and mention action of spring request handler.
  - In request handler take arg @RequestParam("file") CommonsMultipartFile file.
  - File data will be accessible using file.getBytes(). Process that data.
- Hibernate blob handling
  - Create table with binary data as BLOB type.
  - In entity class map column with @Column and @Lob to byte[].
- File download
  - In spring request handler take HttpServletResponse as arg and return type void.
  - resp.setContentType(" application/octet-stream");
  - resp.getOutputStream().write(byteArray);



# Spring Application Context

- One spring application can have multiple application contexts.
  - While creating new application context, we must specify parent application context.
  - If a bean is not resolved by child application context, it will ask parent application context to resolve it.
  - Thus child context can access beans of parent; however reverse is not true.



# Spring Web Application Context

- Each spring web application needs at least one WebApplicationContext.
- In simple spring MVC web application, the dispatcher servlet is responsible for creating spring webapplicationcontext.
  - This context is responsible for MVC as well as dependency injection.
- If spring is to be used for DI along with any existing MVC framework like struts or JSF, we cannot use dispatcher servlet.
  - Use ServletContextListener (named as "ContextLoaderListener") is registered in web.xml, which is responsible for creating spring webapplicationcontext.
  - This webapplicationcontext will handle DI and struts/JSF controller will handle MVC.
- In typical Spring MVC applications we can have ContextLoaderListener to create "root" webapplicationcontext and one/more dispatcher servlets for managing MVC.
- The dispatcher servlets also create their own webapplicationcontext, which are child(s) of "root" webapplicationcontext.

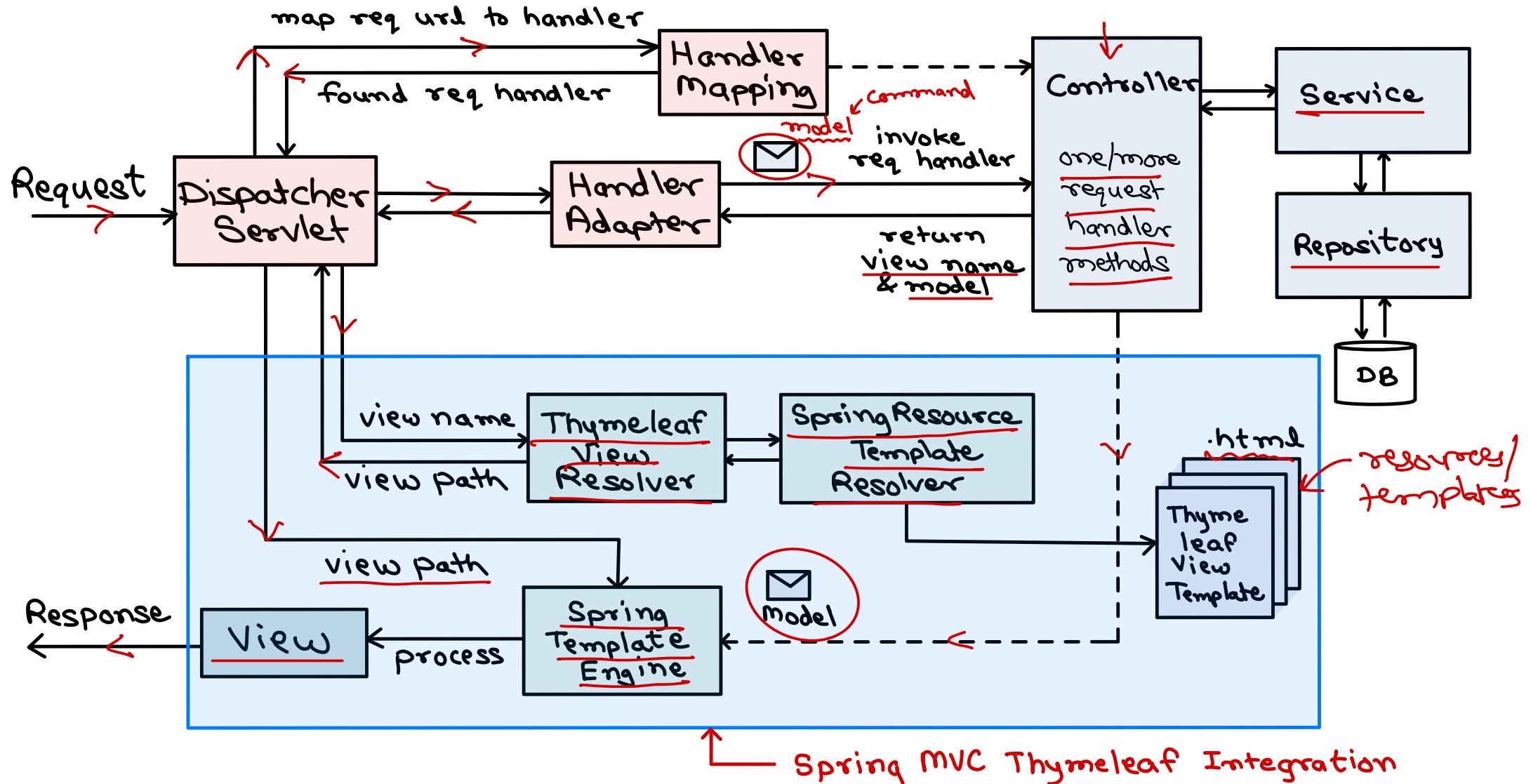


# Spring WebMVC Security

- Authentication
  - Checking user/client credentials.
  - Who am I?
- Authorization:
  - Checking whether user/client is allowed to access the resource.
  - Am I authorized?
- Role Based Security:
  - Each user is assigned ROLE. A user may be in multiple ROLES.
  - Each ROLE is allowed access for certain resources & denied access for certain resources.



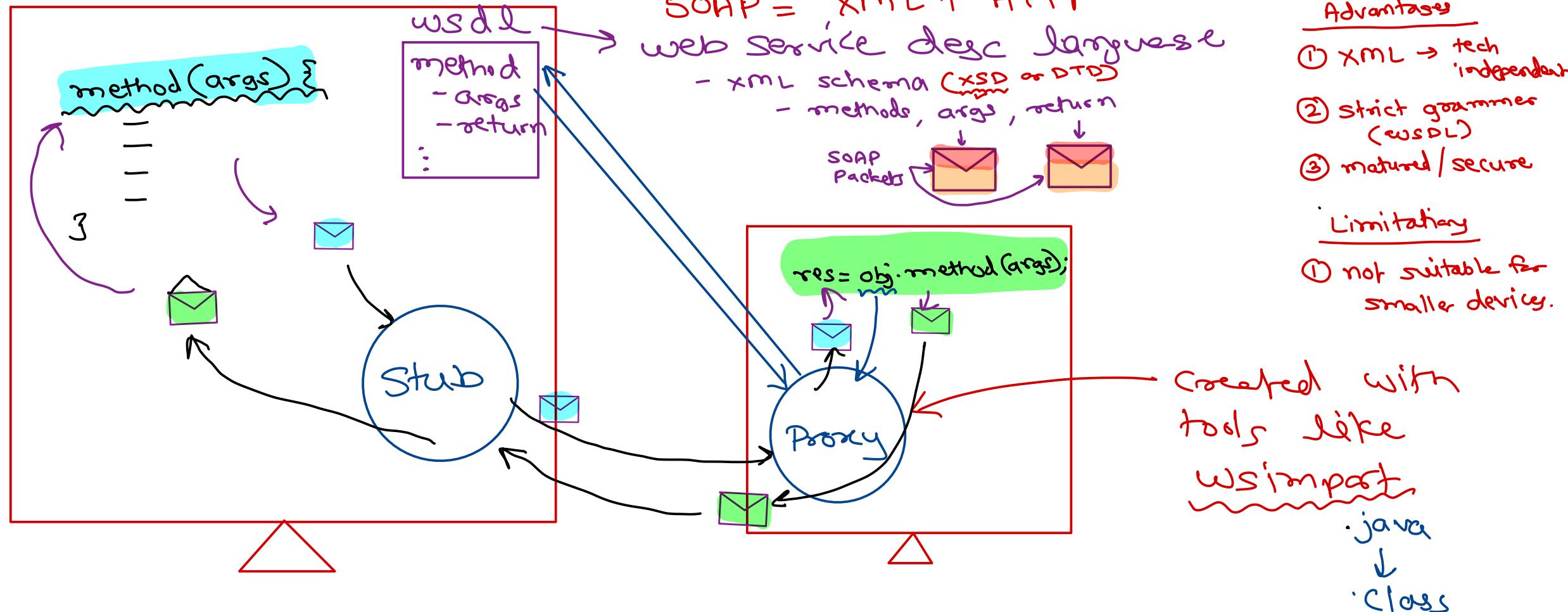
# Spring Web MVC + Thymeleaf



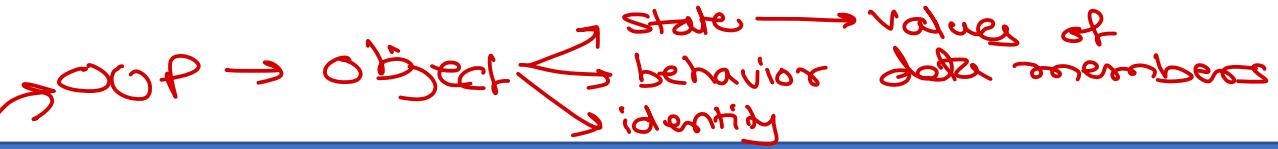
# REST services



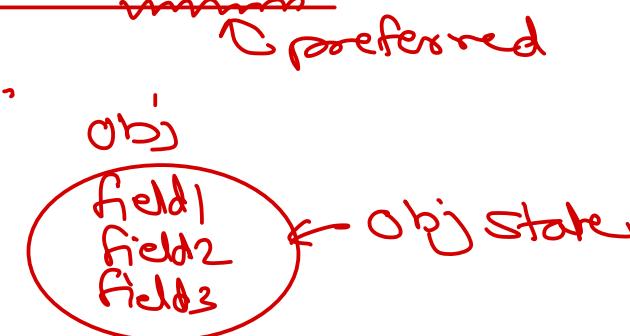
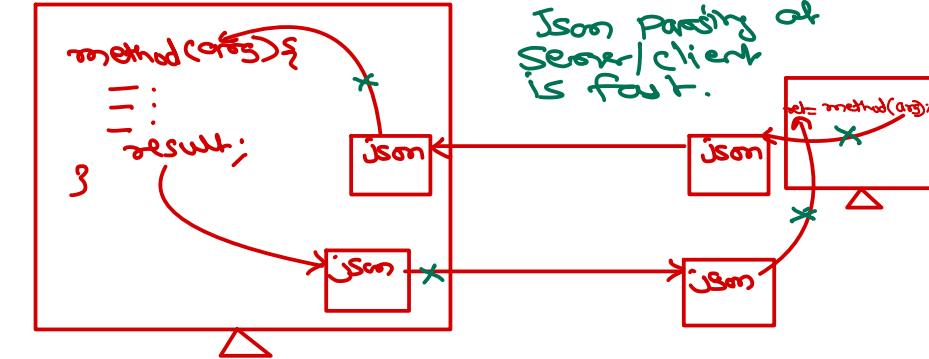
# XML based services - SOAP (Simple Object Access Protocol)



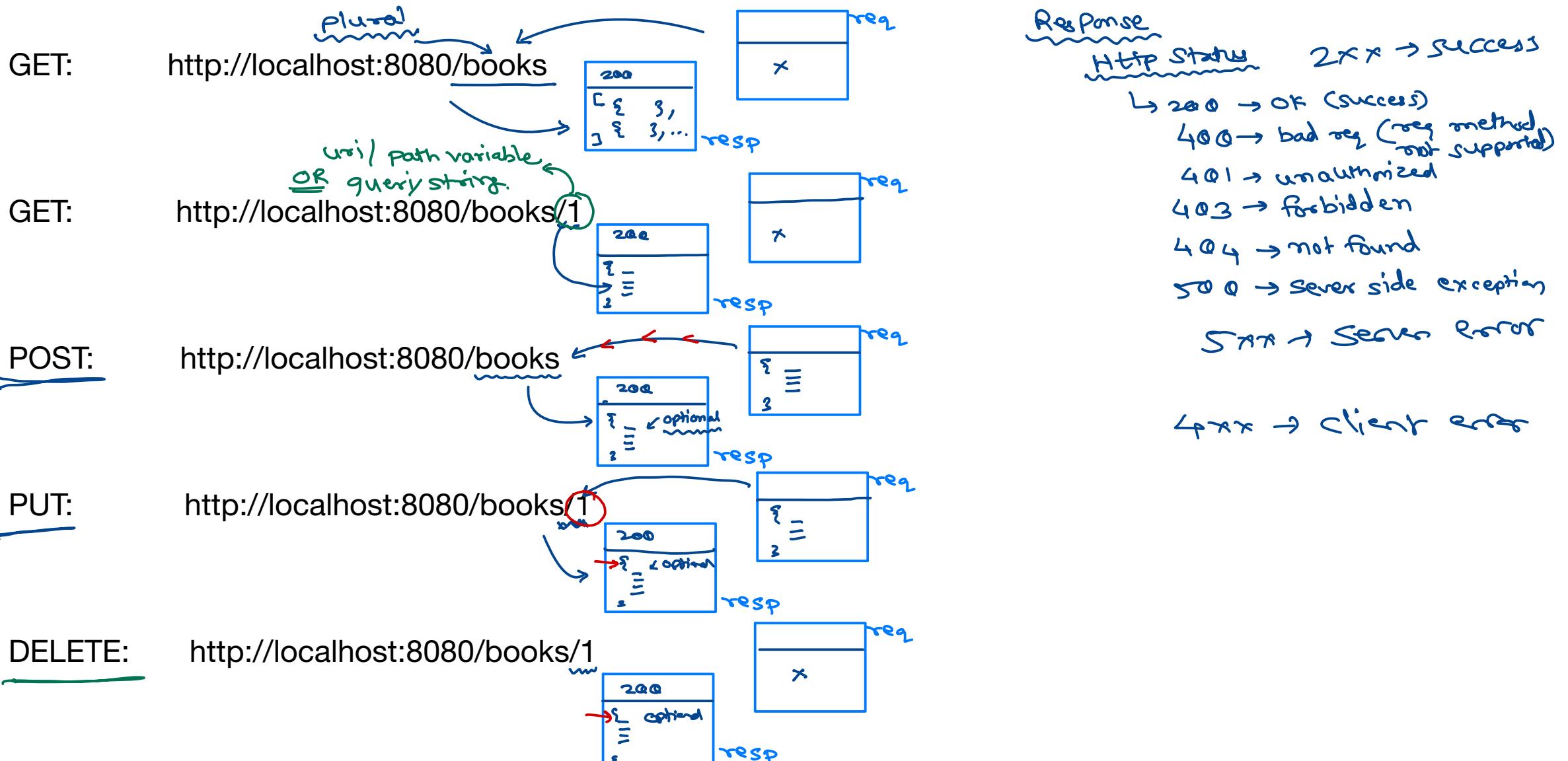
# REST services



- REpresentation State Transfer
- Protocol to invoke web services from any client.
  - Client can use any platform/language.
    - Java
    - .Net
    - PHP
    - C/C++
- REST works on top of HTTP protocol.
  - Can be accessed from any device which has internet connection.
  - REST is lightweight (than SOAP) – XML or JSON.
  - Uses HTTP protocol request methods
    - GET: to get records
    - POST: to create new record
    - PUT: to update existing record
    - DELETE: to delete record



# REST Services Convention

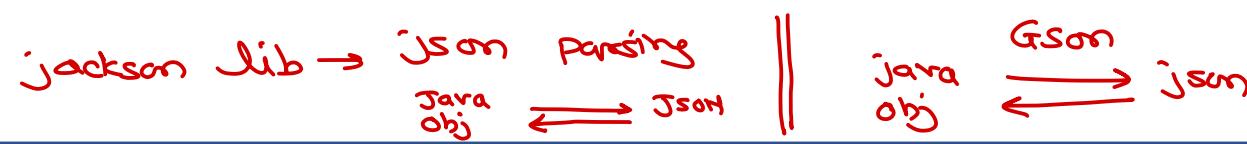


# Spring REST

---



# Spring REST services

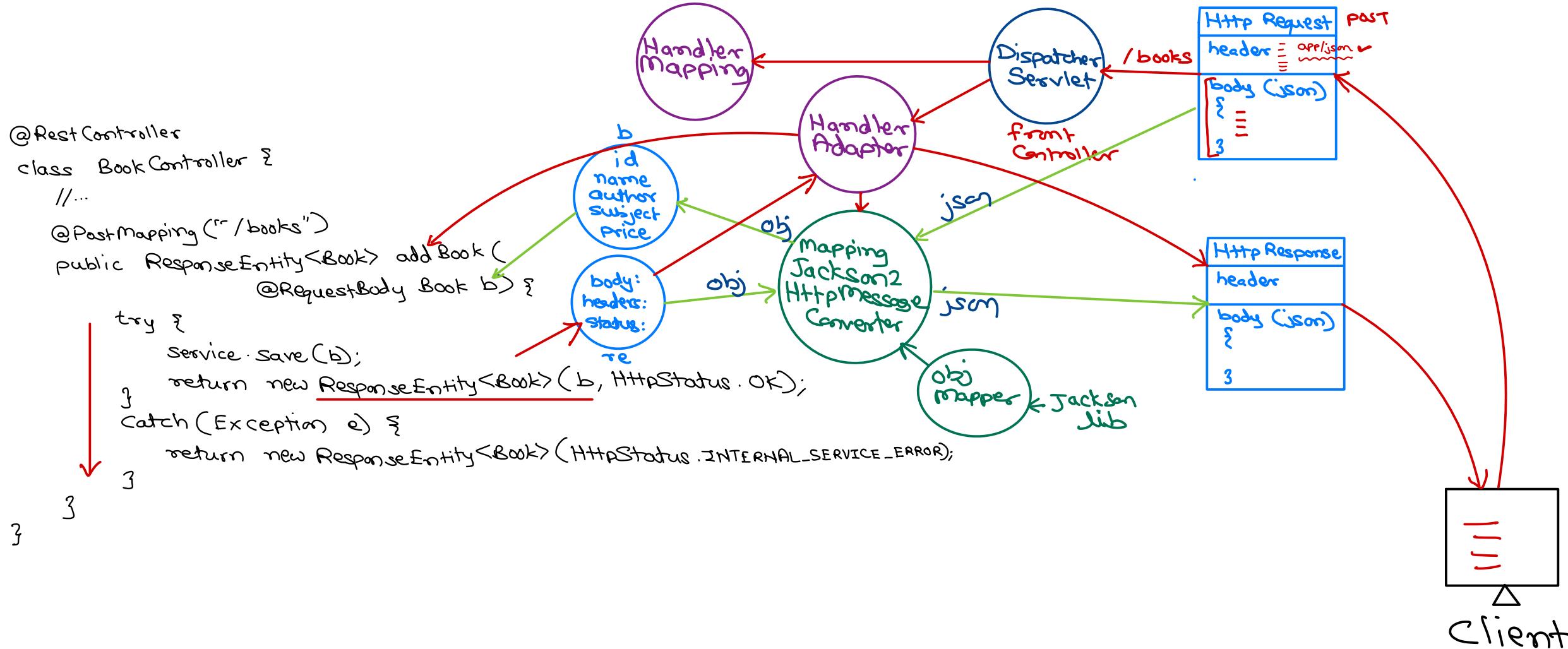


- Based on top of Spring Web MVC.  
↳ Strong conv → for Spring boot  
↳ opinionated default  
↳ web-starter
- Maven dependency: jackson-databind and jackson-databind-xml (if need XML)
- HttpMessageConverter beans:
  - MappingJackson2HttpMessageConverter, MappingJackson2XmlHttpMessageConverter
  - @Override configureMessageConverters() → MVC Config → auto create msg converter beans.
- Using @Controller
  - @GetMapping, @PostMapping, @PutMapping, @DeleteMapping or @RequestMapping
  - @RequestBody, @ResponseBody / ResponseEntity<>
  - @PathVariable, @RequestParam
- Using @RestController
  - @GetMapping, @PostMapping, @PutMapping, @DeleteMapping or @RequestMapping
  - @RequestBody, ResponseEntity<>
  - @PathVariable, @RequestParam
- To manipulate JSON response use @JsonProperty or @JsonIgnore.



# Spring REST services

```
@RestController  
class BookController {  
    ...  
    @PostMapping("/books")  
    public ResponseEntity<Book> addBook(  
        @RequestBody Book b) {  
        try {  
            service.save(b);  
            return new ResponseEntity<Book>(b, HttpStatus.OK);  
        } catch (Exception e) {  
            return new ResponseEntity<Book>(HttpStatus.INTERNAL_SERVER_ERROR);  
        }  
    }  
}
```



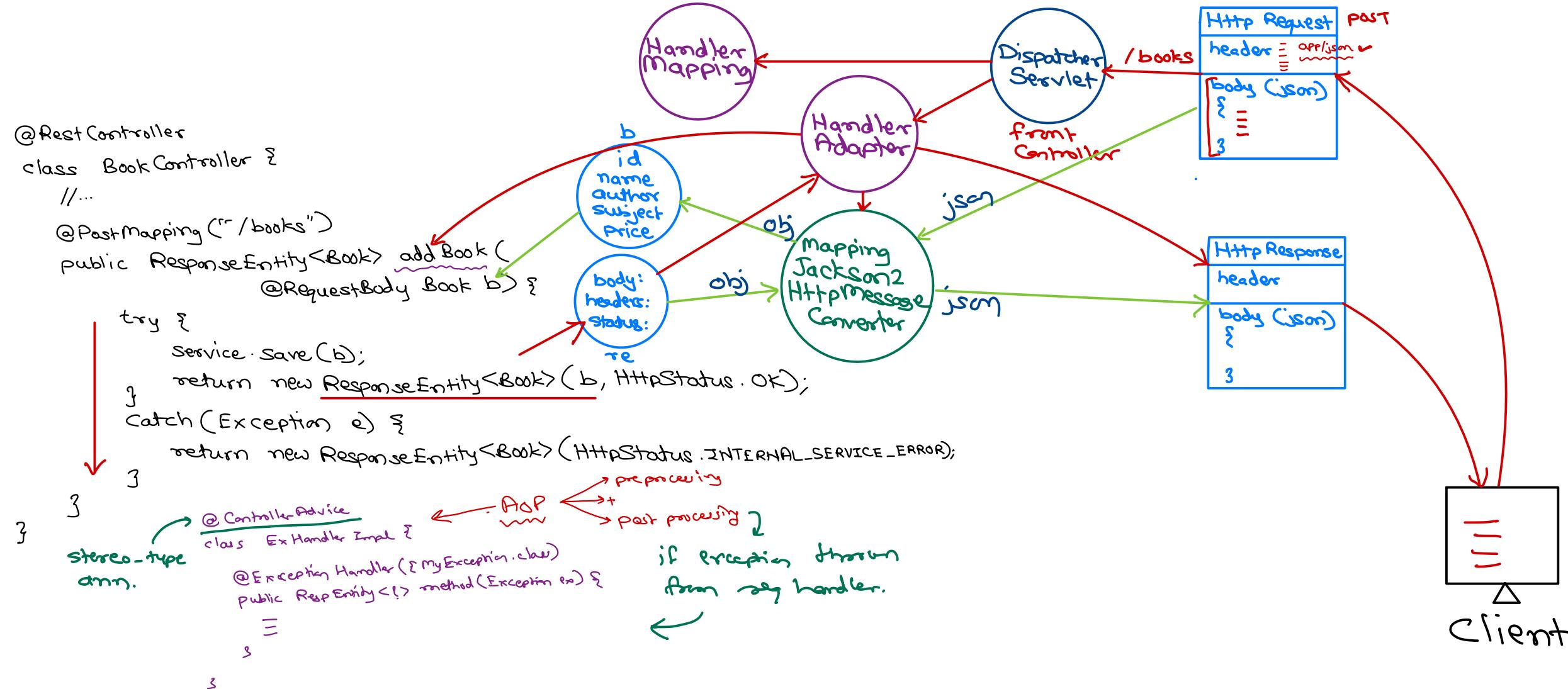
# Spring REST services

```
@RestController  
class BookController {  
    ...  
    @PostMapping("/books")  
    public ResponseEntity<Book> addBook(@RequestBody Book b) {  
        try {  
            service.save(b);  
            return new ResponseEntity<Book>(b, HttpStatus.OK);  
        } catch (Exception e) {  
            return new ResponseEntity<Book>(HttpStatus.INTERNAL_SERVER_ERROR);  
        }  
    }  
}
```

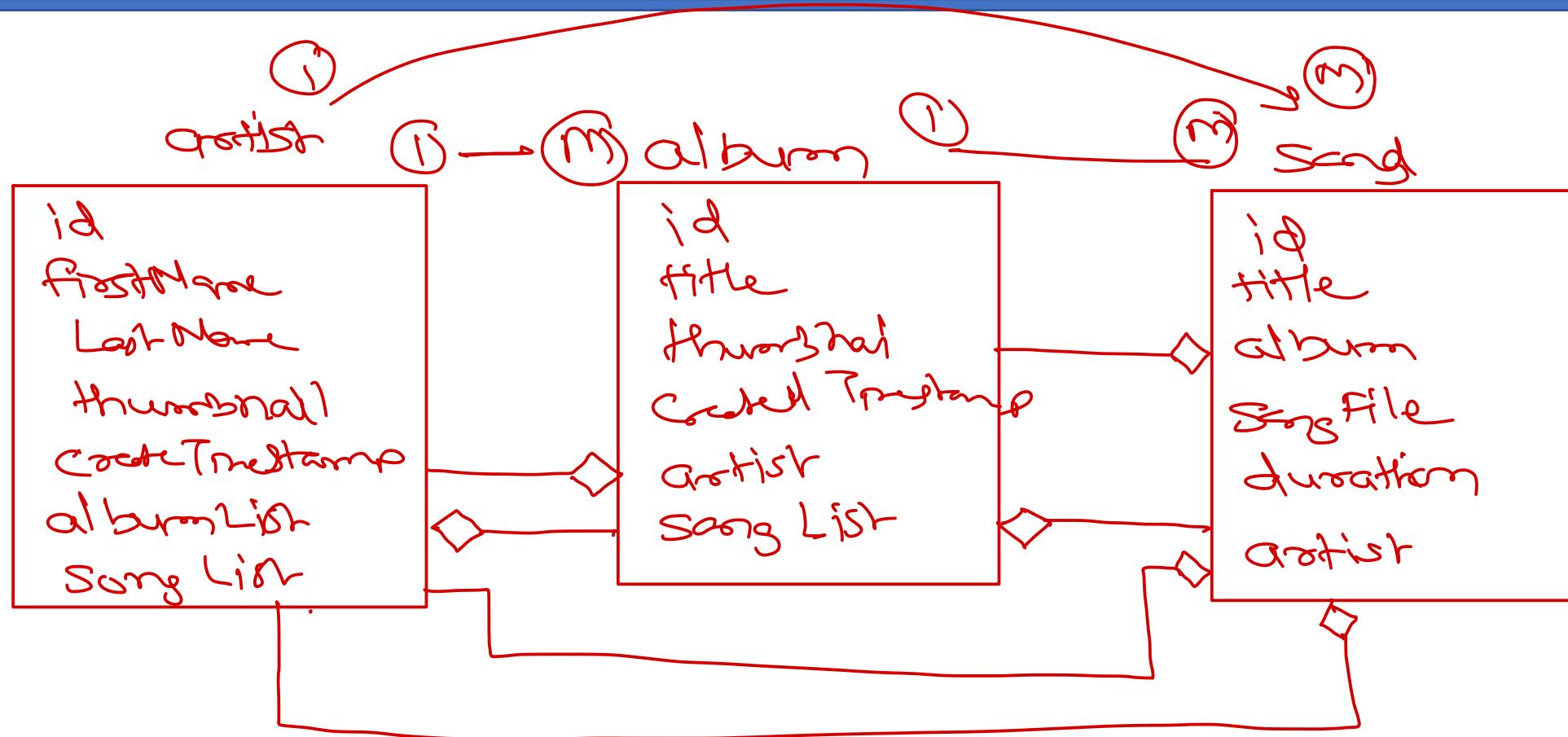
stereotype  
ann.

```
@ControllerAdvice  
class ExHandlerImpl {  
    @ExceptionHandler({myException.class})  
    public ResponseEntity<!> method(Exception ex) {  
        ...  
    }  
}
```

3  
3  
3



# Spring React Example - gaana.com



# Spring REST services

- Spring REST services produce/consume JSON/XML data based on available message converters. Can configure in MVC using `configureMessageConverters()`:
  - `MappingJackson2HttpMessageConverter`, `MappingJackson2XmlHttpMessageConverter`.
- Spring REST services can be invoked by `RestTemplate`.
  - Using this can execute GET, POST, PUT, DELETE, HEAD or OPTIONS requests.
  - `restTemplate.optionsForAllow(url)`;
  - `restTemplate.postForEntity()`, `restTemplate.postForObject()`
  - `restTemplate.getEntity()`, `restTemplate.getForObject()`
  - `restTemplate.put()`, `restTemplate.delete()`
  - `restTemplate.exchange()`
- Spring5 introduced `WebClient` that can be used to invoke REST services.
  - Based on reactive pattern.
  - Can invoke synchronously or asynchronously.



## Rest Services are Stateless

Stateless (do not store client details).

- ① REST is based on HTTP.
- ② Since HTTP is stateless, Rest is stateless.
- ③ In REST services state ongoing (session/cookie) is not advised.
- ④ Each req should be stand alone / self-contained.  
required args should be passed as  
query strings) url var / req body / req header.

## State ongoing in REST app

- ✓ ① session storage (HTML5 local client side).
- ✓ ② database





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

# Java EE

## Agenda

- Spring Boot Deployment
- Spring REST Client
- Spring AOP
- Hibernate Caching
- JPA
- Custom Tags

## Spring Boot Deployment

- Application Build
  - Application config (database config, disk folder config, email config, ...)
  - Maven --> Build --> Package
- Deploy the Application
  - Option 1: On some machine
    - On target machine ensure that Java 11 is installed.
    - Copy the jar on that machine.
    - terminal> java -jar app.jar
  - Option 2: Cloud (linux) machine (e.g. AWS EC2)
    - Install Java 11 on that machine.
    - Copy the jar on that machine.
    - terminal> nohup java -jar app.jar &
      - nohup command block the HANGUP signal.
      - When a terminal is closed, HANGUP signal sent to process, which terminates the process.
      - "&" at the end is async execution of the command i.e. shell will not wait for application completion. The application runs in background.
  - Option 3: Cloud services (e.g. AWS Beanstalk)
  - Option 4: Docker (A light-weight virtual machine)
    - Pre-requisite: Docker is installed.
    - Follow steps below.
  - Option 5: Deploy in Tomcat as web application
    - Follow steps below.

## Deploying JAR package into Docker

- step 1: Create basic maven starter web project.
- step 2: Implement a simple @Controller or @RestController.
- step 3: Create maven jar file using Eclipse/Maven CLI.
- step 4: Create "Dockerfile" into project directory.

```
FROM openjdk:11-jre-slim
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

- step 5: Build docker image
  - On terminal go to project directory
  - cmd> docker build -t sunbeam/hello .
- step 6: Run docker container
  - cmd> docker run -p 8080:8080 sunbeam/hello
- step 7: In browser: http://localhost:8080/index
- step 8: Stop docker container
  - cmd> docker ps
  - cmd> docker container stop id

## Creating/Deploying WAR package into Tomcat

- step 1: Create basic maven starter web project.
- step 2: Implement a simple @Controller or @RestController.
- step 3: Add spring-boot-starter-tomcat jar as provided in pom.xml.
- step 4: Make <packaging>war</packaging> in pom.xml
- step 5: Add <finalName>\${artifactId}</finalName> in pom.xml <build> tag.
- step 6: Update maven project (if eclipse).
- step 7: Entry-point class must be inherited from SpringBootServletInitializer. It creates ServletContext for web application.

```
@SpringBootApplication
public class HelloWebApplication extends SpringBootServletInitializer {
    public static void main(String[] args) {
        SpringApplication.run(HelloWebApplication.class, args);
    }
}
```

- step 8: Create maven war file using Eclipse/Maven CLI.
- step 9: Copy war file into tomcat/webapps directory or upload via tomcat manager.
- step 10: In browser: http://localhost:8080/index

## Spring REST Client

- Any Spring application (console application, web application, rest service application, ...) can invoke REST service (developed in spring boot, express, asp.net web-api, php, ...).
- In micro-services architecture the application is divided into small applications. These are typically REST services. These REST services can call each other for the required data.

## Spring AOP

- Refer slides for the concepts.

## Hibernate Caching

- Refer slides
- <https://www.baeldung.com/hibernate-second-level-cache>

## JPA

- Refer slides

## JSP Custom Tags

- Refer slides
- JSP has mainly two types of tags i.e. classic tags and simple tags.
- Both are inherited from marker interface "JspTag".
- To implement these tags you can use interfaces "Tag" and "SimpleTag" or adapter classes "TagSupport" and "SimpleTagSupport" respectively.

### Steps to implement custom (Simple) tag

- step 0: decide the application, name, attributes and body of the tag.
  - e.g. <my:wish uname="some\_name"/>
  - This tag should print greeting message for the given name.
- step 1: write the tag handler class inherited from "SimpleTagSupport" e.g. WishTag. Also add paramless ctor.
  - In the class write number of fields equal to number of attributes, with names matching to attribute. Also add getter/setter for them. Override doTag() method of the "SimpleTagSupport" class and implement the business + presentation logic in it.

```
public class WishTag extends SimpleTagSupport {  
    private String uname;  
    public WishTag() {  
        this.uname = "";  
    }  
    public void setUname(String uname) {  
        this.uname = uname;  
    }  
    public String getUsername() {  
        return this.uname;  
    }  
    public void doTag() {  
        JspContext ctx = this.getJspContext();  
        JspWriter out = ctx.getOut();  
        out.println("Hello, " + uname);  
    }  
}
```

```

    }
}

```

- step 2

- write tag library descriptor (tld) xml file inside WEB-INF to give complete information about the tag.

```

<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/webjsptaglibrary_2_1.xsd">
    <tlib-version>1.0</tlib-version>
    <short-name>custom</short-name>
    <uri>/WEB-INF/custtags.tld</uri>
    <tag>
        <name>wish</name>
        <tag-class>sub.krd.bkshop.WishTag</tag-class>
        <body-content>empty</body-content>
        <attribute>
            <name>uname</name>
            <required>true</required>
            <rteprvalue>true</rteprvalue>
            <type>java.lang.String</type>
        </attribute>
    </tag>
</taglib>

```

- step3: Use the tag into the JSP page.

```

<%@taglib prefix="my" uri="/WEB-INF/custtags.tld" %>
...
<my:wish uname="\${lb.username}"/>

```

## SimpleTag life cycle

- When page containing custom tag is accessed first time, during translation stage:
  - the tld file is referred (via given prefix in @taglib)
  - from tld used tag is found and validated (syntax)
- For each time tag is used, tag-class (given in .tld file) is loaded and object is created at runtime. Paramless constructor will be called here.
- setJspContext() method will be called by the container and current JSP's PageContext object will be passed into it. This object contains all info needed to process JSP page.
- If tag is child of any other tag, then setParent() method will be called.
- Then container calls setter methods for all attributes used in the JSP file during tag invocation.
- If tag has some body, then setJspBody() method will be called to provide tag body.

- Finally container calls doTag() method of the tag class, which does server side processing and generate html output if any.
- After doTag() is completed, the tag's generated html will be added into page response.
- If not overridden, setJspContext() method of the "SimpleTagSupport" will be called (step 3), which will save the current JspContext so that it can be accessed via call to getJspContext() [usually in doTag() method].

SUNBEAM INFOTECH