**Spring**

* An open-source, app framework to reduce complexity of enterprise app development
* It has layered architecture and loosely coupled modules
* Enables POJO programming that assists in continuous integration and testability
* Supports AOP (Aspect Oriented Programming) and DI (Dependency Injection)
* Spring container creates and manages the life cycle of application objects (components)



**Fig. 1: Spring Architecture**

* Spring container creates objects, wires them together, configures them and manages their complete life cycle. Spring configuration information configures the Spring container. This is provided in form of:
  + **XML** (starting from Spring 2.0 it is a schema-based configuration)

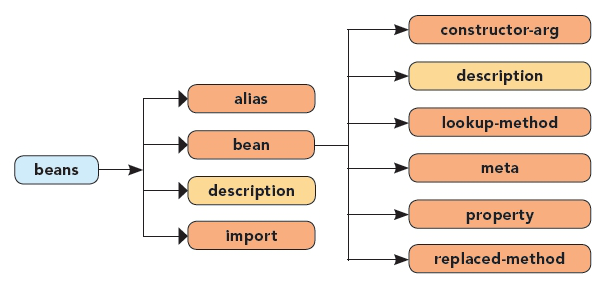
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns=<http://www.springframework.org/schema/beans> xmlns:xsi=<http://www.w3.org/2001/XMLSchema-instance> xsi:schemaLocation="http://www.springframework.org/schema/beans <http://www.springframework.org/schema/beans/spring-beans-2.5.xsd>">

<!-- put configuration details here -->

</beans>

* + **Annotations** (applicable on components)
  + **Java code** (applicable through Java codes)



**Fig. 2: Spring XML configuration elements**

* Instead of putting all bean descriptions in one XML, it’s better to segregate those across several XML files and imported to compose a single unit by using the import element as:

<import resource=”service-layer-config.xml” />

<import resource=”data-layer-config.xml” />

* Spring annotations can be categorised into

* + **Context configuration (core framework annotations)**
    - @Autowired

Applied to fields, setter methods and constructors. Injects object dependency implicitly

* + - @Configuration

Used on classes that define beans. It is an analog for an XML configuration file using Java classes

* + - @ComponentScan

Allow spring to know the packages to scan for annotated components. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation

* + - @Order
    - @Lazy

Used on the component class. By default, all autowired dependencies are created and configured at startup, this will change this behaviour to only when first requested for

* + - @Bean

This is used at method level. The method annotated with this annotation works as the bean ID and it creates and returns the actual bean

* + - @Qualifier

Used along with Autowired for more control on the DI process. Used to avoid confusion that occurs when you create more than one bean of the same type and wire to wire only of them with property

* + - @Required

Applied to bean setter methods. Affected bean must be populated at configuration time with the required property. Otherwise BeanInitializationExeception is thrown

* + - @Value

Used at field, constructor parameter or method parameter level. Inject values from a property file into the bean’s attribute

* + - @Scope
  + **Transaction**
    - @Transactional
  + **Stereotyping**
    - @Component

Indicates the class as a spring component. This marking of the class or bean assists the component scanning mechanism to add it to the application context

* + - @Controller
    - @Repository
    - @Service
  + **Spring MVC**
    - @Controller
    - @InitBinder
    - @ModelAttribute
    - @RequestMapping
    - @RequestParam
    - @SessionAttributes
  + **JMX** (for MBean operations and attributes)
  + **Testing** (for unit tests in Junit 4 style)

**Spring boot annotations**

@SpringBootApplication

- This annotation is placed on the main application class under the root package

- This root package helps for the component scan (@ComponentScan) - thru all the available packages

- This annotation enables following three features:

Use auto configuration in Spring boot application (@EnableAutoConfiguration). This implicitly defines a base search package. Tells Spring boot to start adding beans based on CLASSPATH settings, other beans or various property settings

Use component scan (@ComponentScan)

Able to define extra configuration on its application.class (@Configuration)

@Import({ MyConfig.class, MyAnotherConfig.class })

- used to explicitly import the config classes

@Autowired

- Used for constructor injection

Spring boot autoconfiguration:

For auto configuration, Spring boot looks at

- Frameworks available on the CLASSPATH

- Existing configuration for the application

Based on these, Spring boot provides basic configuration needed to configure the application with these frameworks.

All autoconfiguration logic is implemented in spring-boot-autoconfigure.jar.

Another important file inside spring-boot-autoconfigure.jar is /META-INF/spring.factories.

This file lists all the auto configuration classes that should be enabled under the EnableAutoConfiguration key.

DEBUG logging:

---------------

You can turn on debug logging by adding a simple property value to application.properties.

In the example below, we are turning on Debug level for all logging from org.springframework package (and sub packages).

logging.level.org.springframework: DEBUG

Spring Initializr: A simple tool with UI to create a varied range of spring-boot based applications.

It is accessed thru the link https://start.spring.io/

-------------------------

Spring boot architecture:

-------------------------

The building blocks of Spring boot are:

- Spring Boot Starter Projects

Starters are a set of convenient dependency descriptors that you can include in your application.

For example, if you want to get started using Spring and JPA for database access, just include the spring-boot-starter-data-jpa dependency

in your project. And you are good to go.

- Spring Boot Starter Parent (spring-boot-dependencies is the parent of this)

Spring Boot Dependencies defines the default dependency management for all Spring Boot projects.

If we would want to use a new version of a specific dependency, we can override the version by specifying a new property in the project pom.

- Auto configuration

The most important feature of Spring is DI. DI helps to develop loosely coupled applications. These can easily be unit tested

--------------------

Restart vs Reload

--------------------

The restart technology provided by Spring Boot works by using two classloaders.

Classes that do not change (for example, those from third-party jars) are loaded into a base classloader.

Classes that you are actively developing are loaded into a restart classloader.

When the application is restarted, the restart classloader is thrown away and a new one is created.

This approach means that application restarts are typically much faster than “cold starts”, since the base classloader is already available and populated.

* Dependency Injection in Spring is done in two ways:
  + Constructor injection
  + Setter injection
* IoC containers in Spring
  + BeanFactory
  + ApplicationContext (built over bean factory plus some other features)
* How configuration metadata is provided to container
  + XML based configuration
  + Annotation based configuration
  + Java based configuration
* Bean property values can be

* + value (in case of simple primitive values or String)
  + ref (in case it is an object)
  + Spring configurations can be split into several XMLs using import tag
    - <import resource=”service-layer-config.xml” />
* Five bean scopes are supported by Spring
  + Singleton
  + Prototype
  + Request
  + Session
  + Global-session
* Bean life cycle



* Autowiring has the following modes
  + no (the default)
  + byName
  + byType
  + constructor
  + autodetect

* To turn on annotation based wiring , you must add the following line in beans definition XML as
  + <context: annotation-config />

* Bean initialization types
  + If lazy-init is false (default) bean is loaded when application comes up
  + If true, bean is loaded when first explicit call is made to it
* Main annotations for spring beans



* @Required annotation

* + Is applied to bean property setter methods
  + Should be populated at configuration time with an explicit property value
* @Qualifier

* + To remove ambiguity between beans of same types to be used

* @RequestMapping
  + Class level
  + Method level

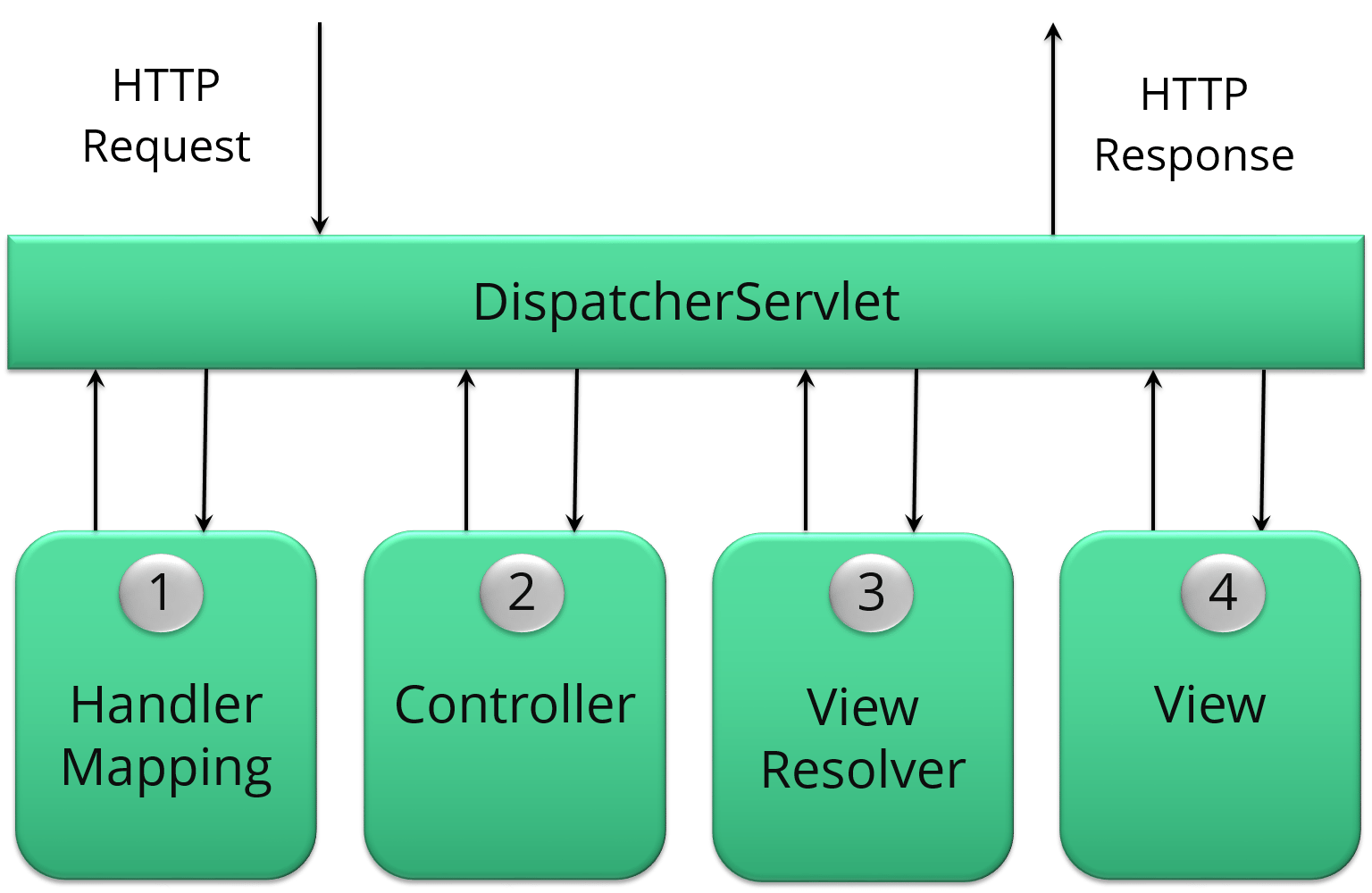
* Spring DAO layer allows you to switch between persistence technologies easily
* Ways by which Hibernate can be accessed using Spring
  + IoC with a hibernate template and callback
  + Extending Hibernate DAO support and applying an AOP interceptor node

* Transaction management types supported by Spring
  + Programmatic TM (extreme flexibility but difficult to maintain)
  + Declarative TM (TM separated from business code)

* Spring AOP

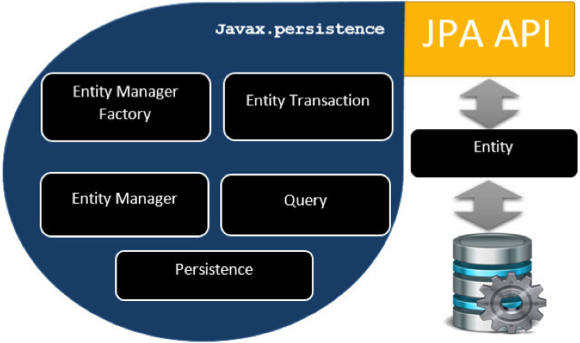
* + Joinpoints
  + Advice (5 types)
  + Aspects
  + Cross-cutting concerns

* Spring web MVC framework



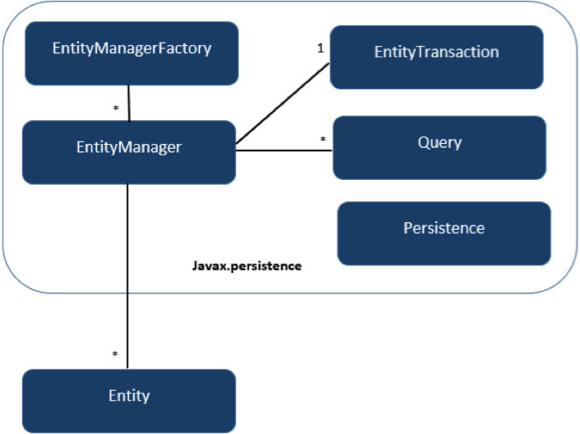
**JPA**

* JPA forms a bridge between object models (Java codes) and relational models (database). First introduced by Oracle with EJB 3.0
* JPA is a specification. Spring data, Hibernate, iBatis are some of its implementations

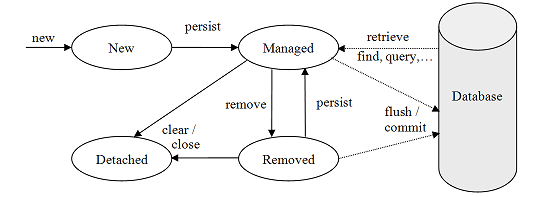


**Class level architecture of JPA**

* The following diagram shows JPA class relationships



* The main concept of JPA is to make a duplicate copy of database in cache memory. While transacting with the database, first it will affect the duplicate data and only when it is committed using entity manager, the changes affect the database
* **Persistence** is the JPA bootstrap class. It contains many static methods to perform important operation in JPA context including methods to obtain **EntityManagerFactory** instance
* The main role of **EntityManagerFactory** instance is to support instantiation of **EntityManager** instances. Instantiation of **EntityManagerFactory** is one-time job, once done it can serve the entire application
* A connection to database is represented by the **EntityManager** instance. It also provides functionality for performing **CRUD** operations on a database
* Operations that modify the content of a database require active transactions. Transactions are managed by an **EntityTransaction** instance obtained from the **EntityManager**
* **EntityManager** also acts as a factory for **Query** instances, that are needed for executing queries on the database
* The basic unit of persistence in JPA is the **Entity** which is nothing but a regular Java class with metadata to describe how its state maps to the database tables. These are POJOs representing fields of a DB table with setters and getters



**Entity object life cycle**

* To use JPA in a Spring project, the **EntityManager** needs to be set up. JPA can now be fully setup with no XML. But when use together, XML mappings can override the values specified in annotations

**Entity:**

* Every entity class should have an **@Entity** marker and an **identifier** field, indicated by **@Id**, that is mapped to the primary key column in the database
* When an entity has a field that references one or more other entities, that field maps to a foreign key column, and is called a relationship field. Other than the identifier field, basic mappings do not need to be annotated, but relationships must be specified by their relationship cardinality
* Defaulting rules in JPA mean that you are not required to specify table names and column names that an entity is mapped to. These can be overridden using additional mapping metadata e.g. **@Table** or **@Column**

**EntityManager:**

* The main API in JPA, that are used to
  + Create new **Entity**
  + Manufacture **Query** to return sets of existing entities
  + Merge in the state of remotely modified entities
  + Delete entities from database
  + And many more
* Two main kinds of EntityManagers are there:
  + Container-managed (uses a standard container that supports JPA Service Provider Interface - SPI))
  + Non-managed (uses Persistence static method to create an instance)
* The basic purpose of EntityManager is to perform CRUD (Create/Read/Update/Delete) operations on entities

**EntityTransaction:**

* JTA transactions are started or committed using the usual container techniques
  + Either by calling the UserTransaction API
  + Or by making use of container-managed transaction demarcation as in Spring
* JPA supports two different kinds of transactions
  + JTA container Transactions
  + Resource local Transactions

* When using local transactions, the transactions must be demarcated manually by invoking on the **EntityTransaction** instance accessed from the **EntityManager**

**Query:**

* Dynamic queries are objects that are created from an **EntityManager** and then executed
* The Query criteria is specified at creation time as Java Persistence Query Language (JPQL) string e.g. “SELECT p from Pet p”
* Before executing a query a number of possible configuration method calls may be made on the query instance to configure it. As for example:

Query q = em.createQuery(“SELECT p from Pet p”);

q.setMaxResults(100);

List results = q.getResultList( );

* A named query is a query that is defined statically and then instantiated and executed at runtime. It can be defined through the **@NamedQuery** annotation on the Entity class
* Named queries are safer than dynamic queries as they get pre-compiled by the persistence implementation

**JP QL:**

* The Java Persistence Query Language is SQL-like but operates over the entities and their mapped persistent attributes instead of the SQL schema
* JP QL queries can return data projections over entity attributes, averting instantiation of the actual entity objects

**Persistence:**

* There is only one unit of JPA configuration needed to get your application started. It is based on the notion of a **Persistence** unit, and configured in a file called **persistence.xml** that must always be placed in the METAINF directory of deployment unit
* Configuration of Persistence differs depending on whether it is being deployed to a managed-container environment or a non-managed one

**HIBERNATE**

* Hibernate is one of the most widely used ORM tools for Java applications. It is a reference implementation of Java Persistence API (JPA)
* Hibernate implicitly provides transaction management
* Important interfaces of Hibernate
  + SessionFactory (initialized by hibernate configuration file – single per application)
  + Session (core interface between Java application layer and hibernate)
  + Transaction (one session can have multiple transactions)
* Three states of an entity beans
  + Transient
  + Persistent
  + Detached

**Isolation Levels**

* Relational databases are transactional. These transactions provide the ACID capabilities of database
  + Atomicity (A)
  + Consistency (C)
  + Isolation (I)
  + Durability (D)

* Atomicity states that all changes to data are performed as a single unit operation and follows ‘ALL or NONE’ principle
* Consistency states that data should be in a consistent state before and after a transaction. For example, if fund is transferred from one account to other, the total value combining both accounts should be the same before and after transaction
* Isolation ensures that the intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized
* After a transaction successfully completes, changes to data persists and are not undone even in the event of a system failure

**Isolation:**

* Isolation gives the querying user the feeling that he owns the database. It does not matter that hundreds or thousands of concurrent users work with the same database and the same schema (or even the same data).
* There are four isolation levels
  + READ UNCOMMITTED (dirty read) Quickest operation
  + READ COMMITTED
  + REPEATABLE READ
  + SERIALIZABLE Slowest operation