**INTERNITY FOUNDATION**

**TASK-18**

**Submitted By:**

**Amisha Singhal**

**Java Batch**

**Spring - Web MVC Framework**

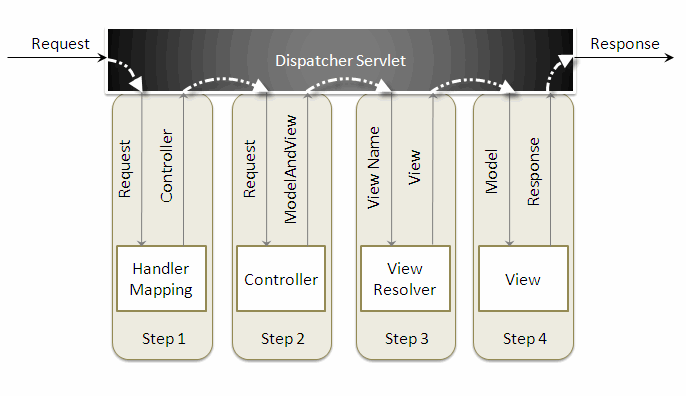
**Ans-** A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection. A Spring MVC provides an elegant solution to use MVC in spring framework by the help of **DispatcherServlet**.**DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* The **Model** encapsulates the application data and in general they will consist of POJO.
* The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

****

**The flow of requests in the spring mvc framework:**



When a request is sent to the spring mvc framework the following sequence of events happen:

* the dispatcherservlet first receives the request.
* the dispatcherservlet consults the handlermapping and invokes the controller associated with the request.
* the controller processes the request by calling the appropriate service methods and returns a modelandview object to the dispatcherservlet . the modelandview object contains the model data and the view name.
* the dispatcherservlet sends the view name to a viewresolver to find the actual view to invoke.
* now, the dispatcherservlet will pass the model object to the view to render the result.
* the view , with the help of the model data, will render the result back to the user.

**Spring - AOP with Spring Framework**

**Ans-** Spring AOP enables Aspect-Oriented Programming in spring applications. In AOP, aspects enable the modularization of concerns such as transaction management, logging or security that cut across multiple types and objects (often termed **crosscutting concerns**).

AOP provides the way to dynamically add the cross-cutting concern before, after or around the actual logic using simple pluggable configurations. It makes easy to maintain code in the present and future as well. We can add/remove concerns without recompiling complete source code simply by changing configuration files.

AOP is mostly used in following cases:

* to provide declarative enterprise services such as declarative transaction management.
* It allows users to implement custom aspects.

**Types of AOP Advices**

1. **Before advice**: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
2. **After returning advice**: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
3. **After throwing advice**: Advice to be executed if a method exits by throwing an exception.
4. **After advice**: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
5. **Around advice:** Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

**Spring - Transaction Management**

**Ans- Spring Transaction Management** is one of the most widely used and important feature of Spring framework. Transaction Management is a trivial task in any enterprise application.Spring provides extensive support for transaction management and help developers to focus more on business logic rather than worrying about the integrity of data in case of any system failures.

The *TransactionDefinition* is the core interface of the transaction support in Spring and it is defined as −

**public interface TransactionDefinition {**

**int getPropagationBehavior();**

**int getIsolationLevel();**

**String getName();**

**int getTimeout();**

**boolean isReadOnly();**

**}**

A transaction strategy is defined by the **org.springframework.transaction.PlatformTransactionManager interface:**

**public interface PlatformTransactionManager {**

**TransactionStatus getTransaction(**

**TransactionDefinition definition) throws TransactionException;**

**void commit(TransactionStatus status) throws TransactionException;**

**void rollback(TransactionStatus status) throws TransactionException;**

**}**

The TransactionDefinition interface specifies:

* ***Isolation*:** The degree to which this transaction is isolated from the work of other transactions
* ***Propagation*:** Typically, all code executed within a transaction scope will run in that transaction.
* ***Timeout*:** How long this transaction runs before timing out and being rolled back automatically by the underlying transaction infrastructure.
* ***Read-only status*:** A read-only transaction can be used when your code reads but does not modify data. Read-only transactions can be a useful optimization in some cases, such as when you are using Hibernate.

The **TransactionStatus interface** provides a simple way for transactional code to control transaction execution and query transaction status. The concepts should be familiar, as they are common to all transaction APIs:

**public interface TransactionStatus extends SavepointManager {**

**boolean isNewTransaction();**

**boolean hasSavepoint();**

**void setRollbackOnly();**

**boolean isRollbackOnly();**

**void flush();**

**boolean isCompleted();**

**}**