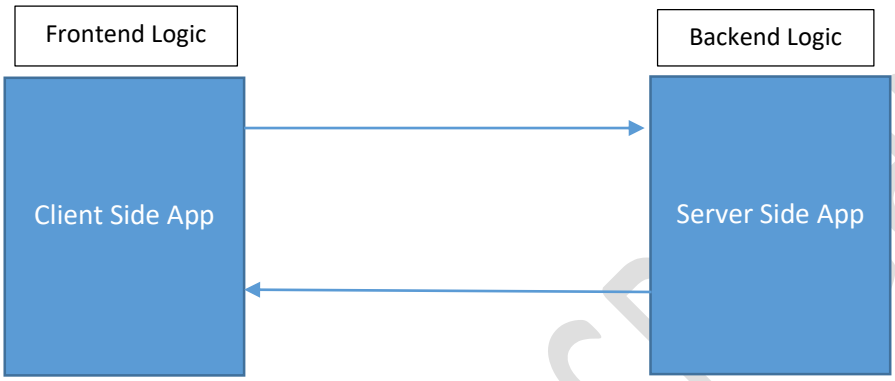
Spring Framework

- Spring is an open source framework which use to develop enterprise application
- Spring is dedicated framework for java development
- Spring provides the infrastructure support for java applications
- With the help of spring framework, we can develop lightweight applications
- Modules of Spring Framework
    - Spring core
    - Spring MVC
    - Spring Rest
    - Spring Boot



- Spring can be used to develop monolithic applications as well as server side application
- An application which contains both the frontend logic and backend logic is called as monolithic application
- In a monolithic application the frontend logic and the backend logic are developing and deployed as a single application
- All though the frontend and backend logic can be developing independently as a client side application and server side application respectively in this case both the applications are deployed independently as two different applications

Spring Core

- Spring core model represent the features of spring framework
- There are two key features of spring framework.
    - IOC
    - DI
- IOC
    - IOC stands for Inversion of Control
    - In spring framework, the programmer is no longer in control or responsible for the object creation and object management.
    - Instead, the spring framework will take the help of a component known as ‘Spring Container’ or ‘IOC Container’
    - This component is responsible to create an object and manage their lifecycle in spring framework
    - The objects created in spring framework are light weight and hence known as ‘bean’
    - In order to create this bean, the spring container or IOC container will take the help of ‘BeanFactory’
    - Even if the spring framework is a control of bean creation, the programmer still responsible to provide the values for bean properties

- The process of providing the property values for a bean is called as bean configuration

```
@Data
public class StudentBean {
	int id;
	String name;
	String email;
}

public class StudentMain{
	public static void main(String [] args) {
		ApplcationContext context = new
ClassPathXmlApplicationContext("StudentConfig.xml");

		StudentBean Student1 = context.getBean(StudentBean.class);

		System.out.println(student1);

		((ClassPathXmlApplicationContext)context).close

	}
}
```

```
<bean class="com.jspider.springcore.bean.StudentBean">
	<property name="id">
		<value>1</value>
	</property>
	<property name="name" value="Ajay"/>
	<property name="email" value="ajay@gmail.com"/>
</bean>
```

- Bean Configuration
  - The values for the bean properties can be provided using two ways
    - Xml based configuration
    - Annotation Based Configuration
- DI
  - DI stands for Dependency Injection
  - The class which depends upon another class for its own bean creation is known as a dependent class
  - The class on which it depends upon is called as dependency class
  - As in the spring framework, the programmer is not responsible to create any objects(beans), hence the spring framework itself has to create a bean of a dependency class and inject it into the bean of the dependent class
  - This feature of the framework is known as 'Dependency Injection'
- XML configuration
  - To configure a bean, we can make use of an Xml Tag '<bean>', in which we can provide the values for the bean properties using '<property>' & '<value>' tag
  - For this we need to create an xml file to provide the configuration
- Dependencies Required for Spring Core
  - Spring core
  - Spring context
  - Project Lombok [optional]
- The xml file used for bean configuration purpose has to be created directly under the source folder (src/main/java)

- To provide bean configuration, we first need to include the schema definition which provide us '<beans>' tag inside which multiple bean tags can be define
- Note
  - The bean configuration can be done with the help of all augmented constructor as well
  - In that case we need to use the '<contructor-arg>' tag in place of property tag
  - To get an all argument constructor for class, we can use '@AllArgsConstructor' which is present in the Lombok package
  - Similarly, we can get a non-parameterized constructor for a class with the help of '@NoArgsConstructor' present in the Lombok package
- **getBean()**
  - it is nonstatic method present in AupplicationContext interface
  - this method is used to obtain and achieve the bean of required class
  - this method can be used in 2 ways that is by passing class as the argument or bypassing a string, wich is considered as the bean name, as the argument
  - if a class is given as the argument to be given then the method will return the bean for that specific class itself
  - but if the bean name is given as the argument(string) then the method will return the object of the supermost class 'Object' in this case we need to use the required class as the cast operator in order to downcast the object return by the method to the required class type
- One To One

```
package com.jspider.springcore.bean;

import lombok.AllArgsConstructor;
import lombok.NoArgsConstructor;

@AllArgsConstructor
@NoArgsConstructor
public class AdharBean {

	int id;
	String adharNo;

	@Override
	public String toString() {
		return "AdharBean [id=" + id + ", adharNo=" + adharNo + "]";
	}
}
package com.jspider.springcore.bean;

import lombok.Data;

@Data
public class PersonBean {

	int id;
	String name;
	AdharBean adhar;
}
```

- One To Many
- Many To One

- Many To Many
- Annotation Configuration
    - In spring Framework, in order to provide the bean configuration through annotations we can make use of the following annotations
        - @Bean
        - @Value
        - @Component
        - @ComponentScan
        - @Autowired
    - @Bean
        - This annotation is used to mark a logic (method) as the required bean configuration
        - This annotation will create a bean based on the value returned by the logic (method)
        - This annotation can be included in a class
        - Note - >
            - in annotation configuration the object of ApplicationContext interface has to be initialize with the help of a class 'AnnotationConficApplicationContext'
            - the argument of class accept the class type as an argument in which '@Bean ' will be present

Create Bean Using Annotation using SetterMethod

```
package com.jspider.anotationspringcore.beans;

import lombok.Data;

@Data
public class StudentBean {

	private int id;
	private String name;
	private String email;

}


package com.jspider.anotationspringcore.config;

import org.springframework.context.annotation.Bean;

import com.jspider.anotationspringcore.beans.StudentBean;

public class StudentConfig {

	@Bean
	public StudentBean getStudent() {
		StudentBean student1 = new StudentBean();
		student1.setId(1);
		student1.setName("Ramesh");
		student1.setEmail("ramesh@gmail.com");
		return student1;
	}
}
package com.jspider.anotationspringcore.main;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.jspider.anotationspringcore.beans.StudentBean;
import com.jspider.anotationspringcore.config.StudentConfig;

public class StudentMain {

	public static void main(String[] args) {
		ApplicationContext context = new
AnnotationConfigApplicationContext(StudentConfig.class);
		StudentBean student1 = context.getBean(StudentBean.class);
		System.out.println(student1);
		((AnnotationConfigApplicationContext)context).close();
	}

}
```

```java
package com.jspider.anotationspringcore.beans;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class StudentBean {

	private int id;
	private String name;
	private String email;

}
package com.jspider.anotationspringcore.config;

import org.springframework.context.annotation.Bean;

import com.jspider.anotationspringcore.beans.StudentBean;

public class StudentConfig {

	@Bean
	public StudentBean getStudent() {
		StudentBean student1 = new StudentBean();
		student1.setId(1);
		student1.setName("Ramesh");
		student1.setEmail("ramesh@gmail.com");
		return student1;
	}

	@Bean("Student2")
	public StudentBean getStudent2() {
		return new StudentBean(2,"Mahesh","mahesh@gmail.com");
	}
}
package com.jspider.anotationspringcore.main;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.jspider.anotationspringcore.beans.StudentBean;
import com.jspider.anotationspringcore.config.StudentConfig;

public class StudentMain {

	public static void main(String[] args) {
		ApplicationContext context = new
AnnotationConfigApplicationContext(StudentConfig.class);

		StudentBean student2 = (StudentBean) context.getBean("Student2");
		System.out.println(student2);
		((AnnotationConfigApplicationContext)context).close();
	}

}
```

- @Value
  - This annotation is used to provide the setter level configuration and initialize the value of a property at declaration itself

```
package com.jspider.anotationspringcore.beans;

import org.springframework.beans.factory.annotation.Value;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class EmployeeBean {

	@Value("1")
	private int id;

	@Value("Mahesh")
	private String name;

	@Value("mahesh@gmail.com")
	private String email;

	@Value("Wakad")
	private String Address;

}
package com.jspider.anotationspringcore.config;

import org.springframework.context.annotation.Bean;

import com.jspider.anotationspringcore.beans.EmployeeBean;

public class EmployeeConfig {

	@Bean
	public EmployeeBean getEmployee() {
		return new EmployeeBean();
	}
}
package com.jspider.anotationspringcore.main;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.jspider.anotationspringcore.beans.EmployeeBean;
import com.jspider.anotationspringcore.config.EmployeeConfig;

public class EmployeeMain {

	public static void main(String[] args) {
		ApplicationContext context = new
AnnotationConfigApplicationContext(EmployeeConfig.class);
		EmployeeBean emp1 = context.getBean(EmployeeBean.class);
		System.out.println(emp1);
		((AnnotationConfigApplicationContext)context).close();
	}

}
```

- Note
  - A spring container gives the priority to the setter configuration over the constructor configuration

- If a bean is configured using all argument constructor but if the bean property is annotated with @Value, then the spring container will override the constructor values to the setter value

- @Component
  - This annotation is used to mark a class to be eligible for the process of bean creation
  - It means if class is marked as @Component, then it is automatically considered by the spring container for bean creation
- @ComponentScan
  - This annotation is used to mark a class to be able to scan the classes or packages for the @Component annotation
- @Autowired
  - This annotation is used to mark a property or a constructor for dependency injection
  - If this annotation given to property, then the setter injection will be performing
  - If this annotation is given to the constructor (Argument Constructor required), then constructor injector will be performed

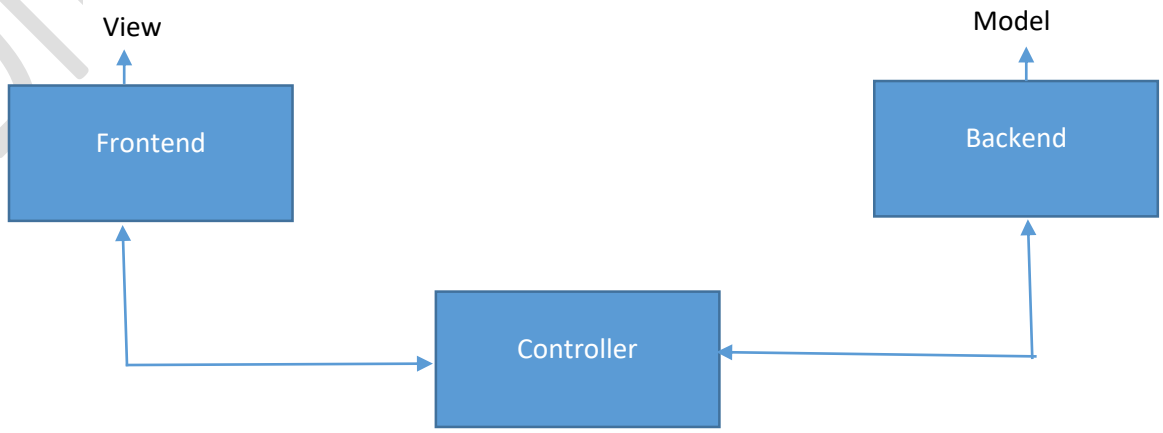
```
package com.jspider.anotationspringcore.beans;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import lombok.Data;

@Data
@Component
public class AdharBean {
	@Value("1")
	private int id;
	@Value("123456789")
	private String adharNo;

}
package com.jspider.anotationspringcore.beans;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;

import lombok.Data;

@Data
public class PersonBean {

	@Value("1")
	private int id;
	@Value("Mahesh")
	private String name;

	@Autowired
	private AdharBean adharBean;
}
```

```java
package com.jspider.anotationspringcore.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;

import com.jspider.anotationspringcore.beans.PersonBean;

@ComponentScan (basePackages = "com.jspider.anotationspringcore") //give here package name

public class PersonConfig {

	@Bean
	public PersonBean getPerson()
	{
		return new PersonBean();
	}

}
package com.jspider.anotationspringcore.main;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.jspider.anotationspringcore.beans.PersonBean;
import com.jspider.anotationspringcore.config.PersonConfig;

public class PersonMain {

	public static void main(String[] args) {
		ApplicationContext context = new
AnnotationConfigApplicationContext(PersonConfig.class);
		PersonBean person1 = context.getBean(PersonBean.class);
		System.out.println(person1);
		((AnnotationConfigApplicationContext)context).close();
	}

}
```

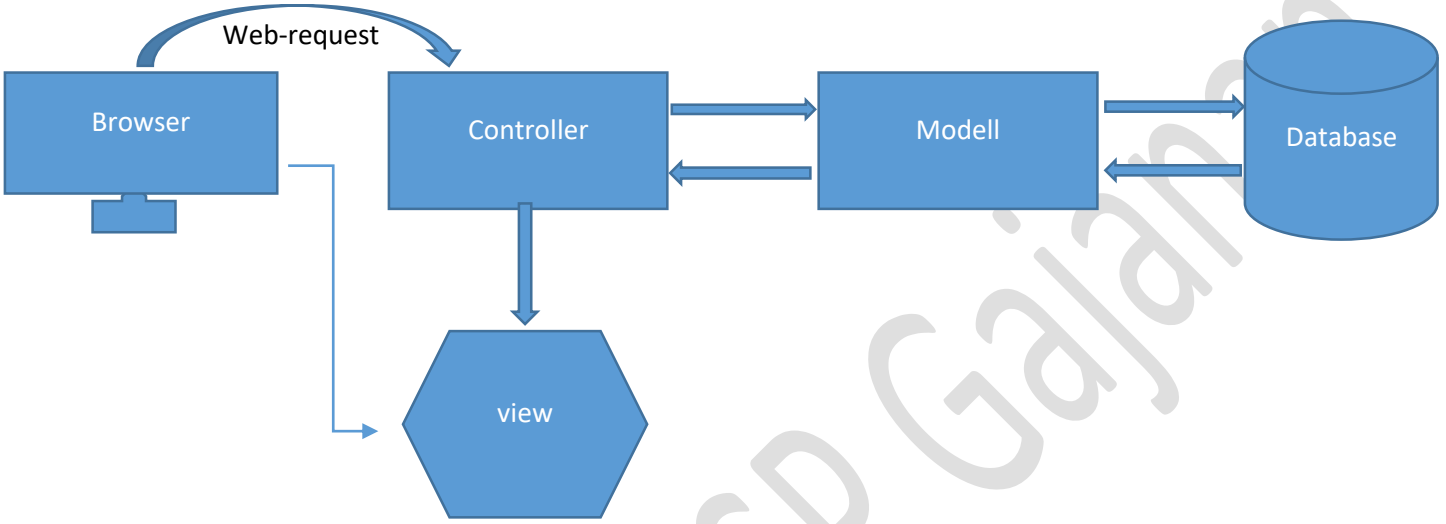
Difference Between Setter Injection and constructor Injection

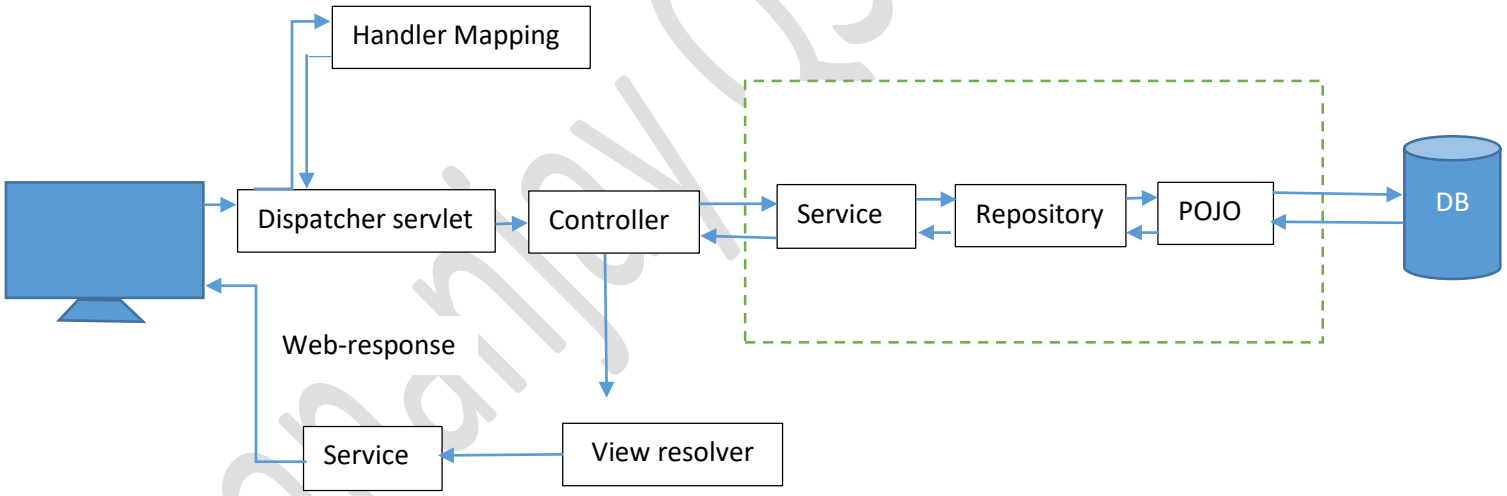
Setter Injection	Constructor Injection
• Setter injection is priorities by the spring container, over the constructor injection • The Autowired injection is given to the properties • Initializing partial properties of the beam is allowed in case of setter injection • If property is not initialize then depending upon its datatype, the default value is considered	• The constructor injection is not priorities if the setter injection exists • The Autowired properties given to the constructor (parametrized Constructor) • Initialing the partial argument of the constructor is not allowed • If an argument is not provided then the constructor cannot execute

- Spring MVC
  - MVC stands for **M**odel **V**iew **C**ontroller
  - MVC is an architectural design pattern
  - It provides architectural support for monolithic application
  - In MVC architecture there are three main components
    - Model
    - View
    - Controller
- Model
  - The model holds the backend logic for the application which is responsible to perform the required business logic & communicate with database
  - After execution the model returns the model data
- View
  - A view holds the frontend logic of the application
  - It contains all the required view pages (JSP or HTML) which can be given as web response
  - The web responses are given as the view pages that contain the model data (whenever required)
- Controller
  - The controller is responsible to accept the web request & analyze it and desired which logic from the model has to be executed
  - After receiving the model
  - data, it will identify the view page that is to be given as web response along with the model data if required
  - we can say that controller acts as a mediator between the model (backend logic) & view (frontend logic)



- MVC Architecture
  - The MVC architecture helps us to manage the flow of data within the application
  - It also determines how the request will be handled in the application and how will be response provided





- Dispatcher Servlet
  - The dispatched servlet component is configured in the web.xml
  - Even if the entry point for all the request coming to the application, is web.xml, it will be now forward into the dispatched servlet it means, now all the incoming web request will be received by the dispatched servlet
  - The dispatcher servlet will take the help of handler mapping component in order to identify which controller is responsible to handle that specific request

    - The dispatcher servlet is also containing the configuration for the view resolver component which helps it to identify and locate the specific view pages that are supposed to be given as the web response
- Controller Layer
    - The controller layer is responsible to analyze the incoming web request based on the url pattern and the mapping method
    - Once the request is analyzing, the controller decide which logic from the service layer has to executed
    - After the logic from service layer is done executing, controller receives the model data from it
    - Now the controller identifies or defines that which view page has to be given has to response
    - Hence the controller has to send the model data to the identified view page located with the help of dispatcher servlet which internally refers to the view resolver
    - Then the identified view page is given as the web response along with the model data
- Note
    - In spring MVC a model is classified into 3 layers that is service, repository, POJO
- Service
    - The service layer is responsible to perform the validation on the incoming and outgoing data
    - It is also responsible to decide which logic from repository layer should be executed and then return the model data back to the controller
- Repository
    - The repository layer it is responsible to perform all the executional logic on the entity class objects from the POJO layer
    - We may say that repository layer will include the hibernate logic
- POJO
    - POJO stand for **P**lane **O**ld **J**ava **O**bject
    - The POJO layer holds all the entity classes required in the application which will not contain any executional logic
- Note

- The controller, service, repository, and POJO layers are defined as packages in the project and the classes under this packages must be annotated as @Controller, @Service, @Repository, @Entity respectively
- @Controller, @Service, @Repository are the combinations of
  - @Target
  - @Retention
  - @Documented
  - @Component

- Note
  - To create an MVC application we need to create a project which is 'Maven-webapp-1.4'
- Configuration For MVC
  - We need to create the layers of model as packages along with the controller layer
  - The dependencies required for MVC must be added in the pom.xml
    - MySQL Connector
    - Hibernate Core
    - Hibernate entity manager
    - Spring MVC
    - Spring orm
    - Servlet api
    - Project Lombok
  - Provide the annotations to the respective layers
  - We need to configure in dispatcher servlet in web.xml

```
<servlet>
          <servlet-name>dispatcher</servlet-name>
          <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
     </servlet>
     <servlet-mapping>
          <servlet-name>dispatcher</servlet-name>
          <url-pattern>/</url-pattern>
     </servlet-mapping>
```

  - To obtain the class path of dispatcher servlet we can make use of 'open type' window which can be access through a shortcut 'ctr+shft+t'

- The forward slash url patterns indicate that all the incoming request will be mapped to dispatcher servlet
- To provide the configuration of handler mapping and view resolver components, we need to create 'dispatcher-servlet.xml' in the same location has that web.xml '(inside the WEB-INF) folder

```
<!-- HandlerMapping Configuration -->
     <context:annotation-config/>
     <context:component-scan
     base-package="com.jspiders.springmvc"/>

     <!-- ViewResolver Configuration -->
     <bean class="org.springframework.web.servlet.
                    view.InternalResourceViewResolver">
          <property name="prefix" value="/WEB-INF/views/"/>
          <property name="suffix" value=".jsp"/>
     </bean>
```

- Inside the web-inf folder we need to create a folder by the name 'views', inside which all the views page all the view pages that are required in the application will be created
- As per the hibernate implementation we need a structure for persistence.xml file, hence that has to be created as well

- @RequestMapping
  - This annotation is used to accept any request in the controller the url pattern in the request is represented as the 'path' attribute where has the method invoke by the request is represented as the method attribute
    - Eg. @RequestMapping(path= "/home ",method=RequestMethod.GET)
- @GetMapping
  - This annotation is used to indicate that controller can accept a request that invokes the GetMethod based on url pattern
- @PostMapping
  - This annotation used to indicate that controller can accept request that invoke the post method based on url pattern
- @RequestParam
  - This annotation is used to fetch the value of a parameter, based on its name, which is present in the web request

- Usually this annotation is accessed in the controller layer

- ModelMap
  - The object of this class is an implicit object which is already initialize and ready to use

- addAttribute()
  - it is a nonstatic method present in a model map class
  - it is an overloaded method
  - one of its signature accepts two arguments in which is of type string and its considered as the name/key of the attribute.
  - the second argument is an object and it is considered the value of the attribute
  - this method is used to send the required attributes to the view pages
  - in the view pages can be access/fetched with help of 'getAttribute()' which accepts the name/key of the attribute as the argument and then returns the object of Object class which can be down casted to the required type

- HttpSession
  - HttpSession is an interface whose Object is an implicit object which helps us to perform operations regarding the session attribute

- @SessionAttribute
  - This annotation is used to access the attribute of the session created for a specific object

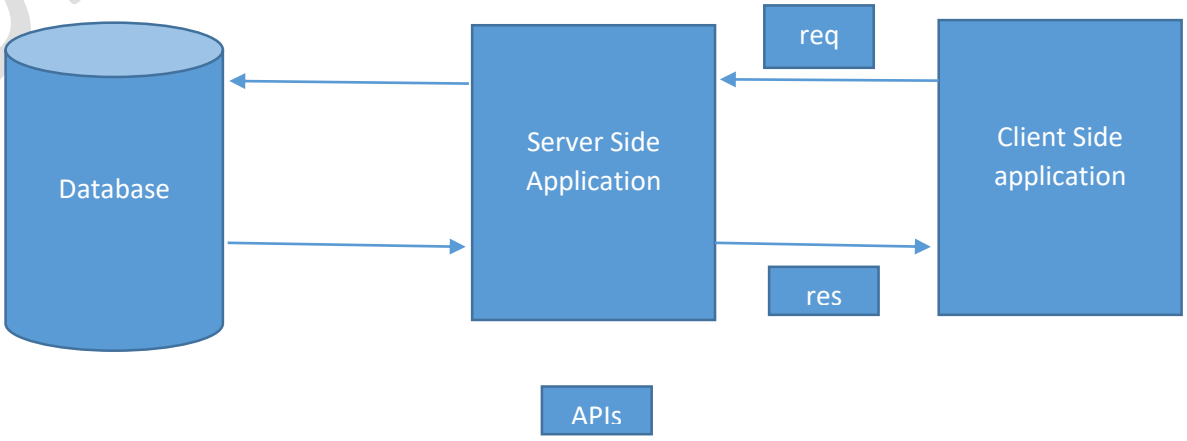
- Methods
  - setAttribute()
    - this nonstatic method from HttpSession is used to initialize the session attribute for a specific object and it also allows us to set the name for the attribute
    - this method accepts 2 arguments out of which the first argument is string which is considered as the name of the session attribute, whereas the second argument is the object for which the session attribute has to be created
  - Invalidate()

- This nonstatic method HttpSession does not accept any argument and used to destroy or terminate the currently active session attribute

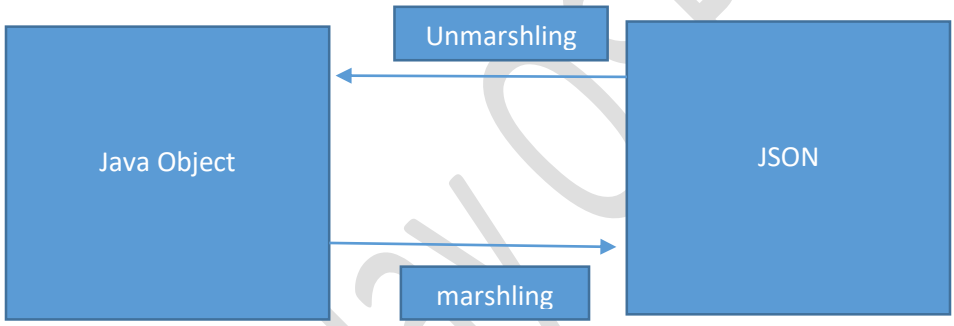
- setMaxInactiveInterval()
  - it is a nonstatic method present in HttpSession it is used to automatically terminate the currently active session after a specific inactive interval
  - the interval is mention as a integer argument in a method
  - the interval is considered in seconds
  - it means after the session created and if there is no activity in the application for the mention interval ,then the session attribute has to be terminated

Spring Rest

- In spring rest module, we develop only the server side application and hence the client side application is developed and deployed independently
- In this scenario the user will not communicate directly with the server side application. Instead the user will now communicate with only the client side application
- The user data which is accepted and displayed in the client side application will be stored at retrieved through the database application only
- The server side application will now act as a mediator between the client side application and the database application
- Moreover, each feature from server side application will independently communicate between the client side and database application hence we can say that the server side application will consist of multiple APIs



- In the server side application, the application data will exist in the form of java objects whereas the most popular language for client side application is Java Script which is incompatible to understand or operate on java objects
- JavaScript majorly deals with the data in JSON format hence when the client side application sends the data to the server side application, it is set in the form of JSON
- The server side application has to be made capable to accepts the JSON data, convert it into java object and then operate on it
- Similarly, while sending the data from the server side application, the java object has to be converted into JSON format which will be acceptable bye the client side application
- The process of converting java object into JSON is known as **Marshalling** whereas the process og converting the JSON object into Java Object is known as Unmarshaling



- Note –
  - As in this module the data is transfer from one state to another hence the name SpringREST
  - Rest stands for **Re**presentational **S**tate **T**ransfer
- Dispatcher-servlet

```
<!-- HandlerMapping Configuration -->
     <context:annotation-config/>
     <context:component-scan
     base-package="com.jspider.springrest"/>

     <!-- Response Configuration -->
     <mvc:annotation-driven/>
```

-
- Note
    - As view pages are not required in springrest, hence the view resolver component is not required to be configured in the dispatcher-servlet, but the response configuration will be required
- Dependencies required for spring rest
    - In addition to the dependencies in spring mvc, we need two more dependencies in spring rest
        - Jackson databind
        - Jackson core
- Note
    - The above dependencies are required to perform marshaling and unmarshling process
- Unmarshling Configuration
    - If a controller is required to accept JSON data, then it has to be first configured for unmarshling
    - Unmarshling configuration is done by adding the ‘consumes’ attribute to the mapping annotation
        - consumes =MediaType.APPLICATIONTYPE_JSON_VALUE
        - in springframework.http.MediaType
- Marshaling configuration
    - If a controller is required to generate JSON value, then it has to be first configured for the process of marshaling
    - It can be done with the help of produces attribute int the mapping annotation
        - produces =MediaType.APPLICATIONTYPE_JSON_VALUE
- @RestController
    - This annotation used for assigning multiple annotation at once to a controller class
    - This annotation internally contains the following annotations
        - @Taget(value={TYPE})
        - @Controller
        - @ResponseBody
        - @Retention (value = RUNTIME)
        - @Documented

- @ResponseBody
  - This annotation is used to indicate that the annotated controller class is capable of generation an object body as the response
  - Note –
    - In springrest the controller class has to be annotated with @Controller & @ResponseBody or instead it can be annotated as @RestController

- @JsonInclude
  - This annotation is used to control or define that which properties from the object of the annotated class should be included or represented in its respective Json structure
    - Eg. @JsonInclude(JsonInclude.Include.NON_NULL)
- @PathVariable
  - This annotation is used to fetch the variable from the request which is define along with the url pattern in the url path
  - In the mapping annotation, the path variable has to be define inside the curly braces {}

SpringBoot

- SpringBoot can be considered as an open source to which is built over the spring framework
- Springboot makes the development of web applications an microservices, faster and easier
- The springboot is famous and widely use due to three key features as follows
    - Autoconfiguration
    - Opinioned
    - Standalone
- Autoconfiguration
    - The configuration required for the application that were done manually while programmer in the spring framework, are done implicitly(automatically) in the springboot tool
    - Hence we can say springboot provides “convention-over-configuration” approach towards the java development
- Opinionated
    - The springboot tool can we considered as opinionated due to its wide support for the starter dependencies Springboot supports over 50 started spring dependencies and many 3rd party starter dependencies
    - The started dependencies allowed the consideration of default values for certain objects and features in the springboot application
- Standalone
    - A springboot application is created using a project type ‘Spring Starter Project’
    - This projects structure includes ‘apache tomcat ’ as the embedded server
    - Hence springboot application does not depend on any 3rd –party web server application which makes it get level to be deployed on any machine independently
    - This makes springboot a standalone application
- Note
    - Eclipse do not support the springstarter project type hence the project has to be created from an external source ‘spring initializr’

- Apart from that we can use the springTool-Suit IDE which can implicitly refer to the spring initializer source and supports spring starter project creation directly

- @SpringBootApplication
  - This annotation is present by default in the springboot application inside the the default class which conntaints the main method it means the execution of the spring boot application starts from the main method inside the default class
  - In order to run this application as springboot application, @SpringBootApplication annotation is mandatory
- run()
  - it is a static method from spring application class which is used to run spring application from the specified source
  - it accepts the default class of the springboot application as an argument and returns the running object of ApplicationContext