accordingly:

```
createNote(state, action) {
state.push(action.payload)
}
```

Changing the importance of notes could be implemented using the same principle, by making an asynchronous method call to the server and then dispatching an appropriate action.

UNIT – IV

Java Web Development

Web development is known as website development or web application development. The web development creates, maintains, and updates web development applications using a browser. This web development requires web designing, backend programming, and database management. The development process requires software technology.

Web development creates web applications using servers. We can use a web server or machine server like a CPU. The Web server or virtual server requires web application using technology. Web development requires server-side programming language or technology. Mostly Java, PHP, and other server-side languages require for web development.

Java web development creates a server-side website and web application. The majority of Java web apps do not execute on the server directly. A web container on the server hosts Java web applications.

For Java web applications, the container acts as a runtime environment. What the Java Virtual Machine is for locally running Java applications, the container is for Java web applications. JVM is used to run the container itself.

Java distinguishes between two types of containers: web and Java EE. Additional functionality, such as server load distribution, can be supported by a container. A web container supports Java servlets and JSP (JavaServer Pages). In Java technology, Tomcat is a common web container.

A web container is usually a minimal need for web frameworks. GWT, Struts, JavaServer Faces, and the Spring framework are common Java web frameworks. Servlets are at the heart of most modern Java web frameworks.

Functions of Java Web Development

Java web development creates applications and websites using static and dynamic resources. The static resource refers to HTML pages with images, and a dynamic resource refers to classes, jars, Servlet, and JSP. Java web development uses several packages, files, and online links. Java web development requires web archive files known as a WAR files.

Java web development works on three main factors. These development factors show below.

- Front-end web development using Java technology.
- Backend web development using Java server technology.
- Database management using Java database driver.

The above three factors create, update, remove, display and operate data or information.

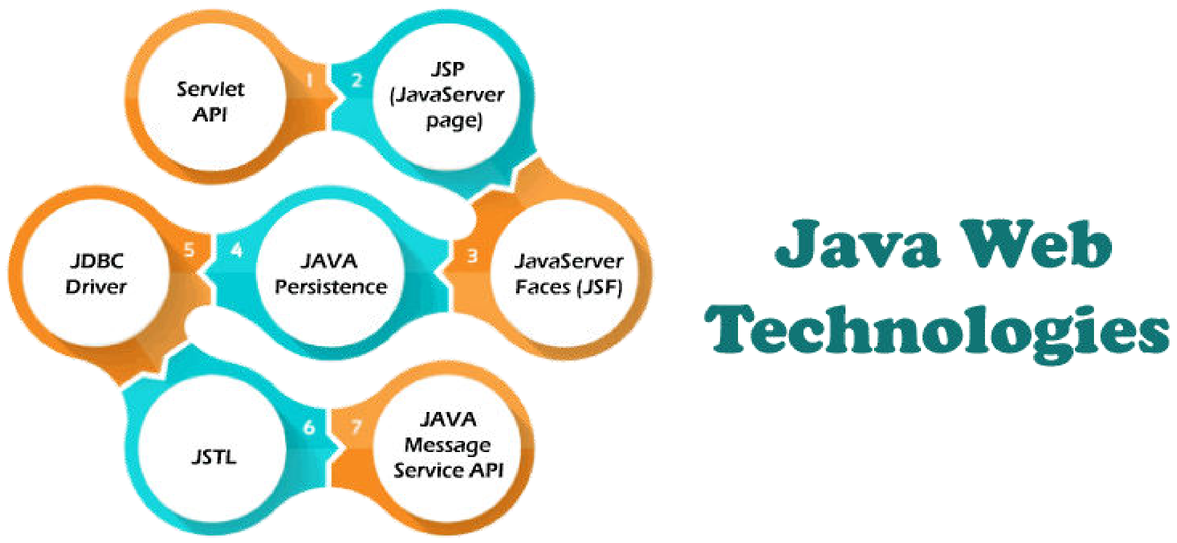
Front-end web development: The front-end technology interacts with the user and Java interface. It helps to insert and submit data. Java web development uses JavaServer Pages or JSP for the front-end form or table.

Backend web development: The backend technology maintains and updates data of the database. Java uses Servlet, spring, and other advanced technology.

Database management handles or fetches data from the database using the Java database driver. The Java technology uses JDBC, Hibernate to handle the database.

Types of the Java Web Technologies

- Servlet API
- JSP (JavaServer page)
- JDBC Driver
- JAVA Persistence
- JavaServer Faces (JSF)
- JSTL
- JAVA Message Service API



Servlet API (JAVA Web application programming interface)

Servlet, filter, filter chain, servlet config, and other interfaces are available in the javax. Servlet package. The capabilities of servers that host apps are increased by using Servlet.

The request-response model is used in web development applications written with Java servlets. From initialization to garbage collection, a servlet has a life cycle.

Servlets are useful for various tasks, including collecting data via web page forms, presenting data from a database or any other third-party source, etc.

Servlets are Java programs that run on a web application and send client requests to databases or servers. After talking with the database, the servlets help process the client's request and provide results.

JSP (JavaServer Page Web application programming technology)

Developers employ JavaServer Pages or JSP technology to quickly produce platform- and server-independent online content. Normally, the developer works on separate Common Gateway Interface files to embed dynamic elements in HTML pages. Java JSP technology can be used, as it has access to the whole Java API family.

The JSP technology pieces code to control web information and moves dynamically. A JSP page comprises static data written in HTML, WML, XML, and other markup languages. Special JSP tags simplify Java code into HTML pages, making web development user-friendly.

The JSP technology allows embedding bits of servlet code in a text-based document. JSP is a popular Java EE technology that allows programmers to create complex dynamic web pages quickly.

JDBC Driver or Java Database Connectivity

JDBC Driver is a connector between database and Java web application. Java database connectivity helps to update and modify data using queries. The jdbc driver is an essential part of Java web development. This driver helps to send data to the database and retrieve data from the database.

Within a Java program, the JDBC driver allows to perform the following tasks:

- Make a data source connection
- To the data source, send queries and update statements
- Displays require data from a database.
- Organize application information.

JDBC is a set of methods and queries for accessing databases written in Java. Clients can use web applications using JDBC drivers to update any information in the database.

JDBC drivers connect to databases in four ways: JDBC-ODBC Bridge Driver, Network Protocol Driver, Native Driver, and Thin Driver.

Persistence API for Java

For web development, the Java Persistence API employs object-relational mapping. This mapping connects a database to an object-oriented model. Java Persistence makes it simple to manage relational data in Java web applications. The Java Persistence API aids in database data management. This API sends data to a database and retrieves data from it regularly.

Large amounts of code, proprietary frameworks, and other files are not required. JPA gives a straightforward technique of database communication. A database is an object-relational approach for interacting with Java web development. JPA is a set of lightweight classes and methods for interacting with databases.

Technology of the JavaServer Faces

JavaServer Faces is called a JSF Technology. This technology provides a framework for developing web-based interfaces. JSF provides a simple model for components in various scripting or markup languages.

The data sources and server-side event handlers are coupled to the User Interface widgets. JSF aids in the creation and maintenance of web applications by minimizing the time and effort required.

- Construct Java web development pages.
- Drop components on a web page by adding component tags to a web page.
- Connect Java web development page components to server-side data.
- Connect component-generated events to application code running on the server.
- Extend the life of server requests by storing and restoring the application state.

Standard Tag Library for JavaServer Pages (JSTL)

The JavaServer Pages Standard Tag Library or JSTL abstracts common functionality of JSP-based applications. We use a single standard set of tags to incorporate tags from various vendors into web applications. This standardization enables the establishment of Java applications on any JSP container. It supports JSTL and increases the tags to optimize during implementation.

JSTL includes iterator and conditional tags for controlling flow. These tags work for manipulating XML documents and tags for internationalization. This JSTL is also used for SQL database access and tags for frequently used functions.

API for Java Message Service

Messaging is a way for software components or apps to communicate with one another. A messaging system is a type of peer-to-peer network. In other words, a messaging client can communicate with and be communicated with by any other client.

Each client establishes a connection with a messaging agent, facilitating the creation, transmission, receipt, and reading of messages.

The Java Message Service (JMS) API provides a strong tool for resolving enterprise computing problems by integrating Java technology and enterprise messaging.

Enterprise messaging enables the secure and flexible sharing of business data. The JMS API extends this by providing a uniform API and provider framework that facilitates the building of portable message-based Java applications.

Special Features of the Java web development

- Java is a mature, versatile, and powerful programming language.
- Additionally, it is popular, which means that tools and assistance for Java web development are readily available.
- Java's platform freedom is one of its strongest characteristics. Java code can be executed on any platform, including a Mac or a Windows computer. On any operating system, we can run a Java web application.
- Java is also capable of running mobile applications on smartphones and tablets.
- Java web development does not require additional effort to design and run web apps across several platforms.
- Java also includes an enormous standard library. This library readily works with common tasks such as input and output, networking, and graphic user interfaces. It provides tools to help web application developers.

Conclusion

Java programming language is easy to handle and programmer's first choice for web development. Java web development has basic rules apart from operating data. This technology does not need an extra operation or advanced programming.

Java web development creates multiple web applications using a single type of code on multiple pages. If we know the working procedure, then JAVA technology develops any application.

JAVA PROGRAMMING BASICS

What is Java?

Java is a high-level, general-purpose, object-oriented, and secure programming language developed by James Gosling at Sun Microsystems, Inc. in 1991. It is formally known as OAK. In 1995, Sun Microsystem changed the name to Java. In 2009, Sun Microsystem takeover by Oracle Corporation.

Editions of Java

Each edition of Java has different capabilities. There are three editions of Java:

- **Java Standard Editions (JSE):** It is used to create programs for a desktop computer.
- **Java Enterprise Edition (JEE):** It is used to create large programs that run on the server and manages heavy traffic and complex transactions.
- **Java Micro Edition (JME):** It is used to develop applications for small devices such as set-top boxes, phone, and appliances.

Types of Java Applications

There are four types of Java applications that can be created using Java programming:

- **Standalone Applications:** Java standalone applications uses GUI components such as AWT, Swing, and JavaFX. These components contain buttons, list, menu, scroll panel, etc. It is also known as desktop alienations.
- **Enterprise Applications:** An application which is distributed in nature is called enterprise applications.
- **Web Applications:** An applications that run on the server is called web applications. We use JSP, Servlet, Spring, and Hibernate technologies for creating web applications.
- **Mobile Applications:** Java ME is a cross-platform to develop mobile applications which run across smartphones. Java is a platform for App Development in Android.

Java Platform

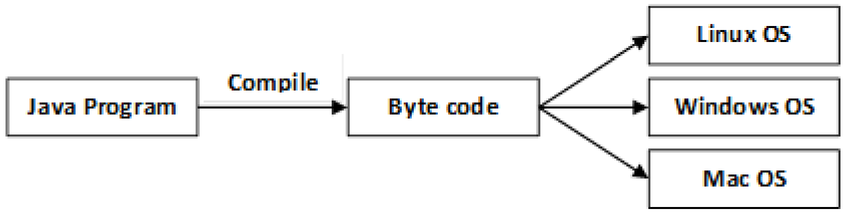
Java Platform is a collection of programs. It helps to develop and run a program written in the Java programming language. Java Platform includes an execution engine, a compiler and set of libraries. Java is a platform-independent language.

Features of Java

- **Simple:** Java is a simple language because its syntax is simple, clean, and easy to understand. Complex and ambiguous concepts of C++ are either eliminated or re-implemented in Java. For example, pointer and operator overloading are not used in Java.
- **Object-Oriented:** In Java, everything is in the form of the object. It means it has some data and behavior. A program must have at least one class and object.
- **Robust:** Java makes an effort to check error at run time and compile time. It uses a

strong memory management system called garbage collector. Exception handling and garbage collection features make it strong.

- **Secure:** Java is a secure programming language because it has no explicit pointer and programs runs in the virtual machine. Java contains a security manager that defines the access of Java classes.
- **Platform-Independent:** Java provides a guarantee that code writes once and run anywhere. This byte code is platform-independent and can be run on any machine.



- **Portable:** Java Byte code can be carried to any platform. No implementation-dependent features. Everything related to storage is predefined, for example, the size of primitive data types.
- **High Performance:** Java is an interpreted language. Java enables high performance with the use of the Just-In-Time compiler.
- **Distributed:** Java also has networking facilities. It is designed for the distributed environment of the internet because it supports TCP/IP protocol. It can run over the internet. EJB and RMI are used to create a distributed system.
- **Multi-threaded:** Java also supports multi-threading. It means to handle more than one job a time.

OOPs (Object Oriented Programming System)

Object-oriented programming is a way of solving a complex problem by breaking them into a small sub-problem. An object is a real-world entity. It is easier to develop a program by using an object. In OOPs, we create programs using class and object in a structured manner.

Class: A class is a template or blueprint or prototype that defines data members and methods of an object. An object is the instance of the class. We can define a class by using the class keyword.

Object: An object is a real-world entity that can be identified distinctly. For example, a desk, a circle can be considered as objects. An object has a unique behavior, identity, and state. Data fields with their current values represent the state of an object (also known as its properties or attributes).

Abstraction: An abstraction is a method of hiding irrelevant information from the user. For example, the driver only knows how to drive a car; there is no need to know how does the car run. We can make a class abstract by using the keyword abstract. In Java, we use abstract class and interface to achieve abstraction.

Encapsulation: An encapsulation is the process of binding data and functions into a single unit. A class is an example of encapsulation. In Java, Java bean is a fully encapsulated class.

Inheritance: Inheritance is the mechanism in which one class acquire all the features of another class. We can achieve inheritance by using the extends keyword. It facilitates the reusability of the code.

Polymorphism: The polymorphism is the ability to appear in many forms. In other words, single action in different ways. For example, a boy in the classroom behaves like a student, in house behaves like a son. There are two types of polymorphism: run time polymorphism and compile-time polymorphism.

Java Variables

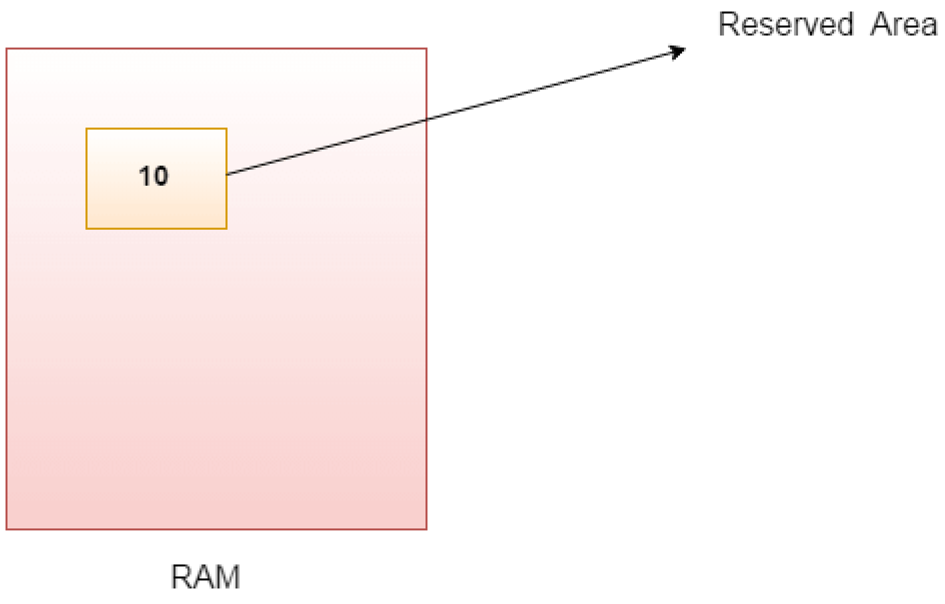
A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in Java: primitive and non-primitive.

Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

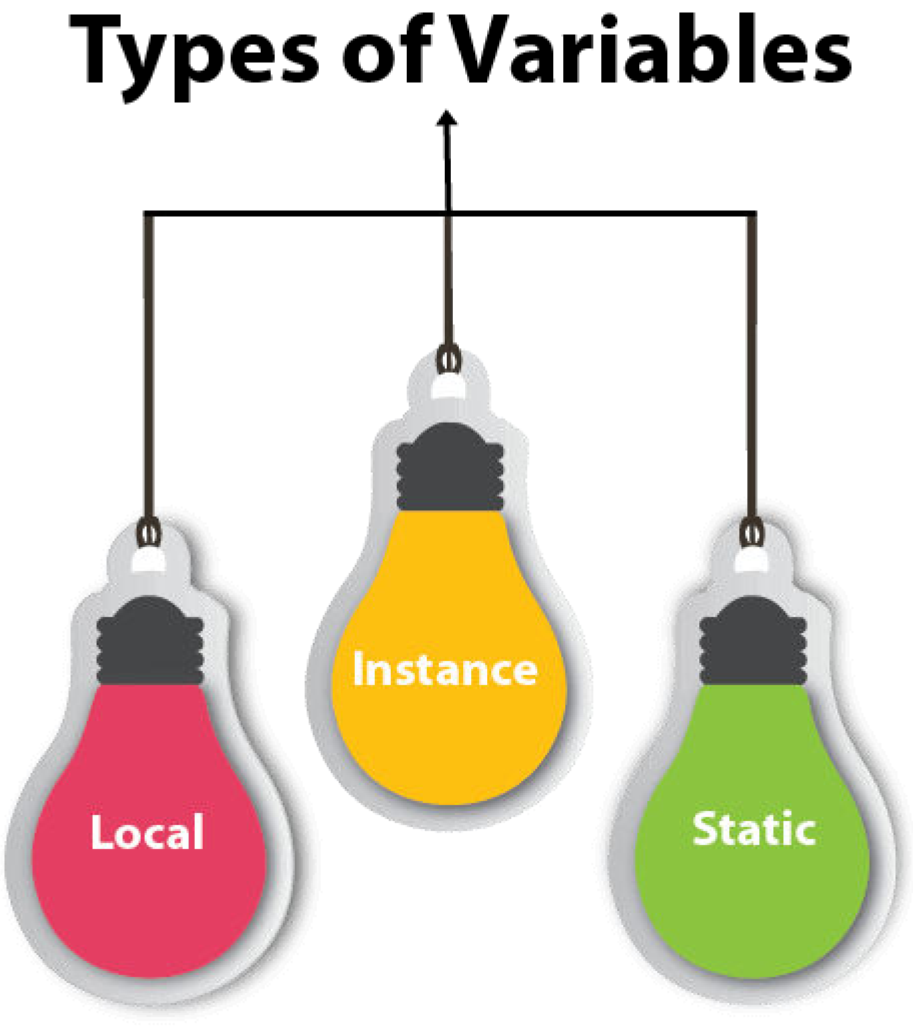


1. int data=50;//Here data is variable

Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable



1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example to understand the types of variables in java

```
public class A
{
    static int m=100;//static variable
    void method()
    {
        int n=90;//local variable
    }
    public static void main(String args[])
    {
        int data=50;//instance variable
    }
}//end of class
```

Java Variable Example: Add Two Numbers

```
public class Simple{
public static void main(String[] args){
int a=10;
int b=10;
int c=a+b;
System.out.println(c);
}
}
```

Output:

```
20
```

Java Variable Example: Widening

```
public class Simple{
public static void main(String[] args){
int a=10;
float f=a;
System.out.println(a);
System.out.println(f);
}}
```

Output:

```
10
```

```
10.0
```

Java Variable Example: Narrowing (Typecasting)

```
public class Simple{
public static void main(String[] args){
float f=10.5f;
//int a=f;//Compile time error
int a=(int)f;
System.out.println(f);
System.out.println(a);
}}
```

Output:

```
10.5
10
```

Java Variable Example: Overflow

```
class Simple{
public static void main(String[] args){
//Overflow
int a=130;
byte b=(byte)a;
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
130
-126
```

Java Variable Example: Adding Lower Type

```
class Simple{
public static void main(String[] args){
byte a=10;
byte b=10;
//byte c=a+b;//Compile Time Error: because a+b=20 will be int
byte c=(byte)(a+b);
System.out.println(c);
}}
```

Output:

```
20
```

Java OOPs Concepts

3. Difference between Object-oriented and Object-based programming language.

In this page, we will learn about the basics of OOPs. Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Smalltalk is considered the first truly object-oriented programming language.

The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

OOPs (Object-Oriented Programming System)

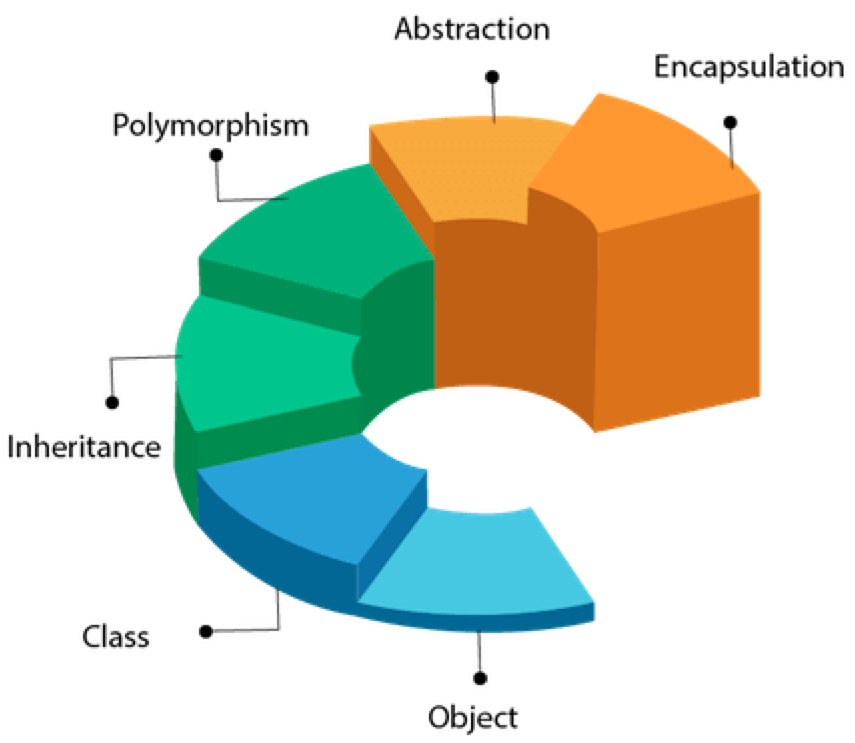
Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- Coupling
- Cohesion
- Association
- Aggregation
- Composition

OOPs (Object-Oriented Programming System)



Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

Advantage of OOPs over Procedure-oriented programming language

1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.

2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

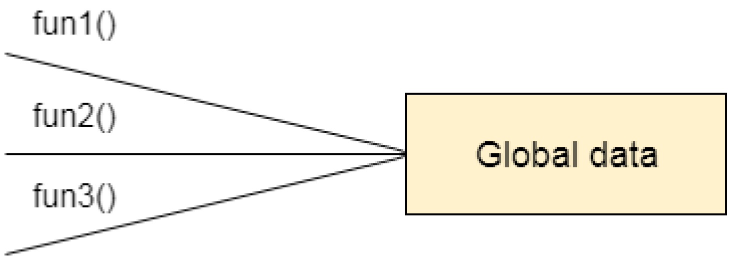


Figure: Data Representation in Procedure-Oriented Programming

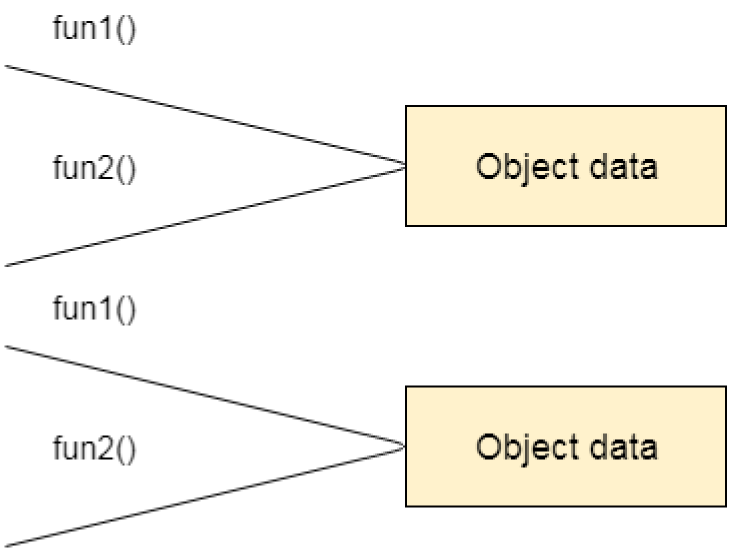


Figure: Data Representation in Object-Oriented Programming

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

What is the difference between an object-oriented programming language and object-based programming language?

Object-based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object-based programming languages.

MVC Architecture in Java

The Model-View-Controller (MVC) is a well-known design pattern in the web development field. It is way to organize our code. It specifies that a program or application shall consist of data model, presentation information and control information. The MVC pattern needs all these components to be separated as different objects.

In this section, we will discuss the MVC Architecture in Java, alongwith its advantages and disadvantages and examples to understand the implementation of MVC in Java.

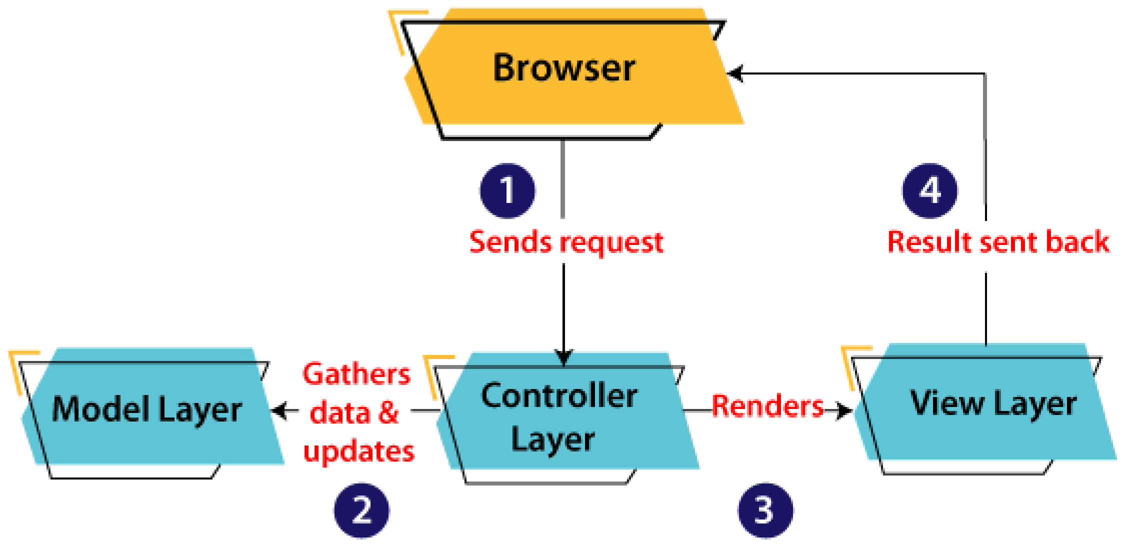
What is MVC architecture in Java?

The model designs based on the MVC architecture follow MVC design pattern. The application logic is separated from the user interface while designing the software using model designs.

The MVC pattern architecture consists of three layers:

- **Model:** It represents the business layer of application. It is an object to carry the data that can also contain the logic to update controller if data is changed.
- **View:** It represents the presentation layer of application. It is used to visualize the data that the model contains.
- **Controller:** It works on both the model and view. It is used to manage the flow of application, i.e. data flow in the model object and to update the view whenever data is changed.

In Java Programming, the Model contains the simple Java classes, the View used to display the data and the Controller contains the servlets. Due to this separation the user requests are processed as follows:



1. A client (browser) sends a request to the controller on the server side, for a page.
2. The controller then calls the model. It gathers the requested data.
3. Then the controller transfers the data retrieved to the view layer.
4. Now the result is sent back to the browser (client) by the view.

Advantages of MVC Architecture

The advantages of MVC architecture are as follows:

- MVC has the feature of scalability that in turn helps the growth of application.
- The components are easy to maintain because there is less dependency.
- A model can be reused by multiple views that provides reusability of code.
- The developers can work with the three layers (Model, View, and Controller) simultaneously.
- Using MVC, the application becomes more understandable.
- Using MVC, each layer is maintained separately therefore we do not require to deal with massive code.
- The extending and testing of application is easier.

Implementation of MVC using Java

To implement MVC pattern in Java, we are required to create the following three classes.

- **Employee Class**, will act as model layer
- **EmployeeView Class**, will act as a view layer
- **EmployeeContoller Class**, will act a controller layer

MVC Architecture Layers

Model Layer

The Model in the MVC design pattern acts as a data layer for the application. It represents the business logic for application and also the state of application. The model object fetch and store the model state in the database. Using the model layer, rules are applied to the data that represents the concepts of application.

Let's consider the following code snippet that creates a which is also the first step to implement MVC pattern.

Employee.java

```
// class that represents model
public class Employee {

    // declaring the variables
     private String EmployeeName;
     private String EmployeeId;
     private String EmployeeDepartment;

    // defining getter and setter methods
     public String getId() {
        return EmployeeId;
     }

     public void setId(String id) {
        this.EmployeeId = id;
     }

     public String getName() {
        return EmployeeName;
     }

     public void setName(String name) {
        this.EmployeeName = name;
     }

     public String getDepartment() {
            return EmployeeDepartment;
        }

     public void setDepartment(String Department) {
            this.EmployeeDepartment = Department;
        }

}
```

The above code simply consists of getter and setter methods to the Employee class.

View Layer

As the name depicts, view represents the visualization of data received from the model. The view layer consists of output of application or user interface. It sends the requested data to the client, that is fetched from model layer by controller.

Let's take an example where we create a view using the EmployeeView class.

EmployeeView.java

```
// class which represents the view
public class EmployeeView {

    // method to display the Employee details
public void printEmployeeDetails (String EmployeeName, String EmployeeId, String EmployeeDepartment){
        System.out.println("Employee Details: ");
        System.out.println("Name: " + EmployeeName);
        System.out.println("Employee ID: " + EmployeeId);
        System.out.println("Employee Department: " + EmployeeDepartment);
    }
}
```

Controller Layer

The controller layer gets the user requests from the view layer and processes them, with the necessary validations. It acts as an interface between Model and View. The requests are then sent to model for data processing. Once they are processed, the data is sent back to the controller and then displayed on the view.

Let's consider the following code snippet that creates the controller using the EmployeeController class.

EmployeeController.java

```
// class which represent the controller
public class EmployeeController {

    // declaring the variables model and view
    private Employee model;
    private EmployeeView view;

    // constructor to initialize
    public EmployeeController(Employee model, EmployeeView view) {
        this.model = model;
        this.view = view;
    }

    // getter and setter methods
     public void setEmployeeName(String name){
       model.setName(name);
     }

     public String getEmployeeName(){
       return model.getName();
     }

     public void setEmployeeId(String id){
       model.setId(id);
     }

     public String getEmployeeId(){
       return model.getId();
     }

     public void setEmployeeDepartment(String Department){
          model.setDepartment(Department);
     }

          public String getEmployeeDepartment(){
            return model.getDepartment();
     }

     // method to update view
     public void updateView() {
         view.printEmployeeDetails(model.getName(), model.getId(), model.getDepart
ment());
     }
   }
```

Main Class Java file

The following example displays the main file to implement the MVC architecture. Here, we are using the MVCMain class.

MVCMain.java

```
// main class
public class MVCMain {
       public static void main(String[] args) {

           // fetching the employee record based on the employee_id from the database
          Employee model = retriveEmployeeFromDatabase();

          // creating a view to write Employee details on console
           EmployeeView view = new EmployeeView();

           EmployeeController controller = new EmployeeController(model, view);
```

```

        controller.updateView();

        //updating the model data
       controller.setEmployeeName("Nirnay");
        System.out.println("\n Employee Details after updating: ");

        controller.updateView();
    }

    private static Employee retriveEmployeeFromDatabase(){
        Employee Employee = new Employee();
       Employee.setName("Anu");
        Employee.setId("11");
       Employee.setDepartment("Salesforce");
        return Employee;
    }
  }
```

The **MVCMain** class fetches the employee data from the method where we have entered the values. Then it pushes those values in the model. After that, it initializes the view (EmployeeView.java). When view is initialized, the Controller (EmployeeController.java) is invoked and bind it to Employee class and EmployeeView class. At last the updateView() method (method of controller) update the employee details to be printed to the console.

Output:

```
Employee Details:
Name: Anu
Employee ID: 11
Employee Department: Salesforce

Employee Details after updating:
Name: Nirnay
Employee ID: 11
Employee Department: Salesforce
```

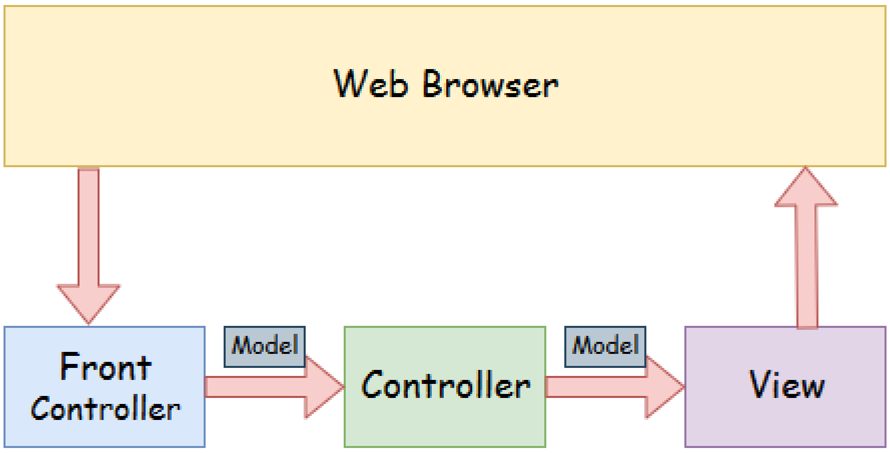
In this way, we have learned about MVC Architecture, significance of each layer and its implementation in Java.

Spring MVC Tutorial

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.

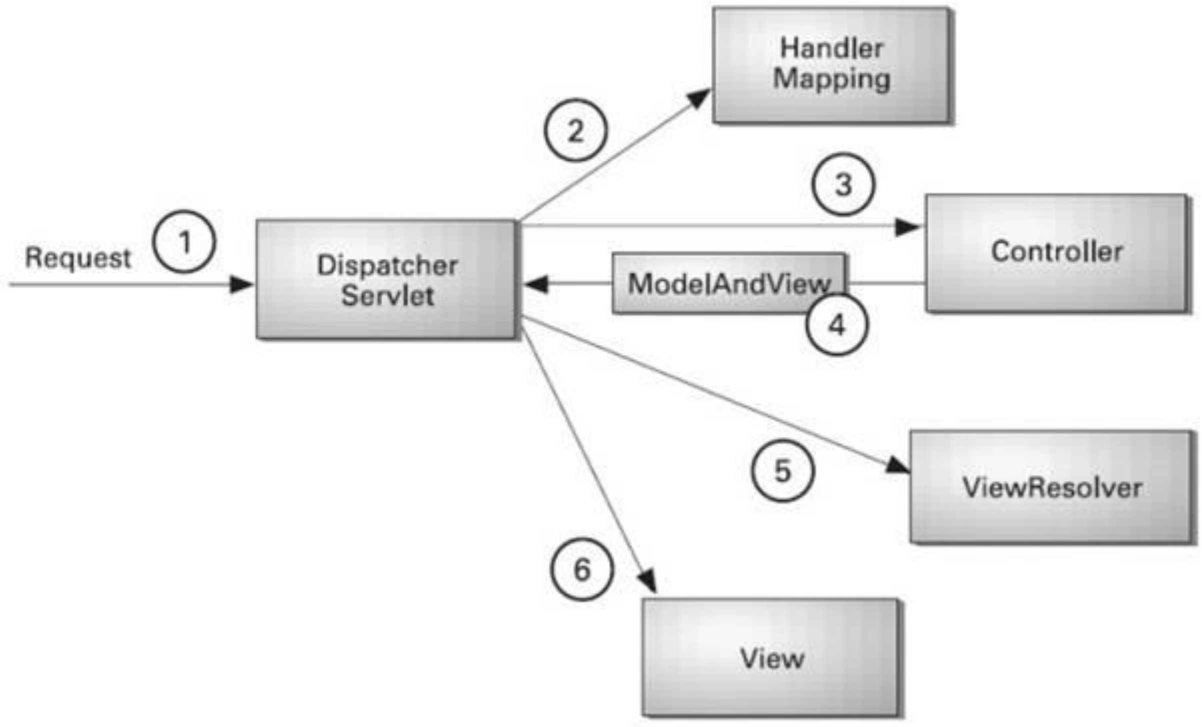
A Spring MVC provides an elegant solution to use MVC in spring framework by the help of **DispatcherServlet**. Here, **DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

Spring Web Model-View-Controller



- **Model** - A model contains the data of the application. A data can be a single object or a collection of objects.
- **Controller** - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
- **View** - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.
- **Front Controller** - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

Understanding the flow of Spring Web MVC



- As displayed in the figure, all the incoming request is intercepted by the DispatcherServlet that works as the front controller.
- The DispatcherServlet gets an entry of handler mapping from the XML file and forwards the request to the controller.
- The controller returns an object of ModelAndView.
- The DispatcherServlet checks the entry of view resolver in the XML file and invokes the specified view component.

Advantages of Spring MVC Framework

Let's see some of the advantages of Spring MVC Framework:-

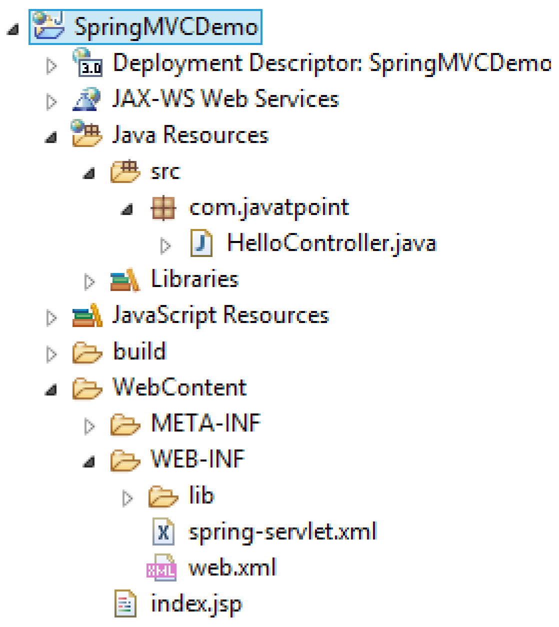
- **Separate roles** - The Spring MVC separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
- **Light-weight** - It uses light-weight servlet container to develop and deploy your application.
- **Powerful Configuration** - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.
- **Rapid development** - The Spring MVC facilitates fast and parallel development.
- **Reusable business code** - Instead of creating new objects, it allows us to use the existing business objects.
- **Easy to test** - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.
- **Flexible Mapping** - It provides the specific annotations that easily redirect the page.

Spring Web MVC Framework Example

Let's see the simple example of a Spring Web MVC framework. The steps are as follows:

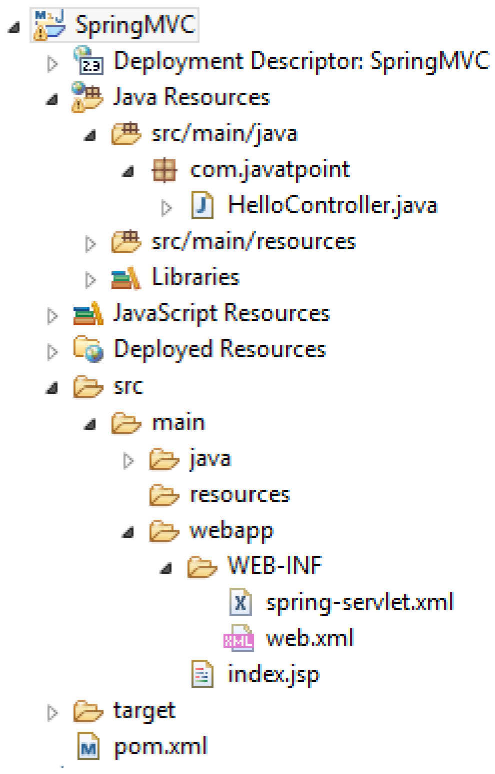
- Load the spring jar files or add dependencies in the case of Maven
- Create the controller class
- Provide the entry of controller in the web.xml file
- Define the bean in the separate XML file
- Display the message in the JSP page
- Start the server and deploy the project

Directory Structure of Spring MVC



```
SpringMVCDemo
  Deployment Descriptor: SpringMVCDemo
  JAX-WS Web Services
  Java Resources
    src
      com.javatpoint
        HelloController.java
    Libraries
  JavaScript Resources
  build
  WebContent
    META-INF
    WEB-INF
      lib
      spring-servlet.xml
      web.xml
    index.jsp
```

Directory Structure of Spring MVC using Maven



```
SpringMVC
  Deployment Descriptor: SpringMVC
  Java Resources
    src/main/java
      com.javatpoint
        HelloController.java
    src/main/resources
    Libraries
  JavaScript Resources
  Deployed Resources
  src
    main
      java
      resources
      webapp
        WEB-INF
          spring-servlet.xml
          web.xml
        index.jsp
  target
  pom.xml
```

Required Jar files or Maven Dependency

To run this example, you need to load:

- Spring Core jar files

- Spring Web jar files
- JSP + JSTL jar files (If you are using any another view technology then load the corresponding jar files).

Download Link: Download all the jar files for spring including JSP and JSTL.

If you are using Maven, you don't need to add jar files. Now, you need to add maven dependency to the pom.xml file.

1. Provide project information and configuration in the pom.xml file.

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javatpoint</groupId>
  <artifactId>SpringMVC</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>SpringMVC Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>

  </dependencies>
  <build>
    <finalName>SpringMVC</finalName>
  </build>
</project>
```

2. Create the controller class

To create the controller class, we are using two annotations @Controller and @RequestMapping.

The @Controller annotation marks this class as Controller.

The @Requestmapping annotation is used to map the class with the specified URL name.

HelloController.java

```
package com.javatpoint;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class HelloController {
@RequestMapping("/")
    public String display()
    {
        return "index";
    }
}
```

3. Provide the entry of controller in the web.xml file

In this xml file, we are specifying the servlet class DispatcherServlet that acts as the front controller in Spring Web MVC. All the incoming request for the html file will be forwarded to the DispatcherServlet.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
   <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

4. Define the bean in the xml file

This is the important configuration file where we need to specify the View components.

The context:component-scan element defines the base-package where DispatcherServlet will search the controller class.

This xml file should be located inside the WEB-INF directory.

spring-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- Provide support for component scanning -->
    <context:component-scan base-package="com.javatpoint" />

    <!--Provide support for conversion, formatting and validation -->
    <mvc:annotation-driven/>

</beans>
```

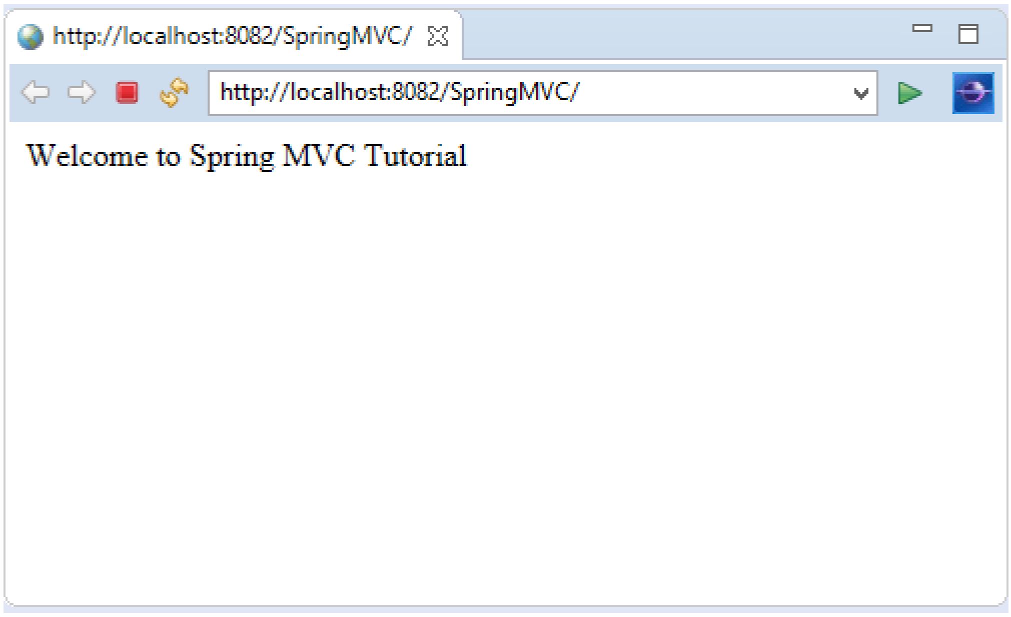
5. Display the message in the JSP page

This is the simple JSP page, displaying the message returned by the Controller.

index.jsp

```
<html>
<body>
<p>Welcome to Spring MVC Tutorial</p>
</body>
</html>
```

Output:



http://localhost:8082/SpringMVC/

Welcome to Spring MVC Tutorial

RESTful API using Spring Framework

Building REST services with Spring

REST has quickly become the de-facto standard for building web services on the web because they're easy to build and easy to consume.

There's a much larger discussion to be had about how REST fits in the world of microservices, but — for this tutorial — let's just look at building RESTful services.

Why REST? REST embraces the precepts of the web, including its architecture, benefits, and everything else. This is no surprise given its author, Roy Fielding, was involved in probably a dozen specs which govern how the web operates.

What benefits? The web and its core protocol, HTTP, provide a stack of features:

- Suitable actions (GET, POST, PUT, DELETE, …)
- Caching
- Redirection and forwarding
- Security (encryption and authentication)

These are all critical factors on building resilient services. But that is not all. The web is built out of lots of tiny specs, hence it's been able to evolve easily, without getting bogged down in "standards wars".

Developers are able to draw upon 3rd party toolkits that implement these diverse specs and instantly have both client and server technology at their fingertips.

By building on top of HTTP, REST APIs provide the means to build:

- Backwards compatible APIs
- Evolvable APIs
- Scaleable services
- Securable services
- A spectrum of stateless to stateful services

What's important to realize is that REST, however ubiquitous, is not a standard, *per se*, but an approach, a style, a set of *constraints* on your architecture that can help you build web-scale systems. In this tutorial we will use the Spring portfolio to build a RESTful service while leveraging the stackless features of REST.

Getting Started

As we work through this tutorial, we'll use Spring Boot. Go to Spring Initializr and add the following dependencies to a project:

- Web
- JPA
- H2

Change the Name to "Payroll" and then choose "Generate Project". A .zip will download. Unzip it. Inside you'll find a simple, Maven-based project including a pom.xml build file (NOTE: You *can* use Gradle. The examples in this tutorial will be Maven-based.)

Spring Boot can work with any IDE. You can use Eclipse, IntelliJ IDEA, Netbeans, etc. The Spring Tool Suite is an open-source, Eclipse-based IDE distribution that provides a superset of the Java EE distribution of Eclipse. It includes features that make working with Spring applications even easier. It is, by no means, required. But consider it if you want that extra **oomph** for your keystrokes. Here's a video demonstrating how to get started with STS and Spring Boot. This is a general introduction to familiarize you with the tools.

The Story so Far…

Let's start off with the simplest thing we can construct. In fact, to make it as simple as possible, we can even leave out the concepts of REST. (Later on, we'll add REST to understand the difference.)

Big picture: We're going to create a simple payroll service that manages the employees of a company. We'll store employee objects in a (H2 in-memory) database, and access them (via something called JPA). Then we'll wrap that with something that will allow access over the internet (called the Spring MVC layer).

The following code defines an Employee in our system.

nonrest/src/main/java/payroll/Employee.java

```
package payroll;

importjava.util.Objects;

importjavax.persistence.Entity;
importjavax.persistence.GeneratedValue;
importjavax.persistence.Id;

@Entity
classEmployee{

private@Id@GeneratedValueLong id;
privateString name;
privateString role;

Employee(){}

Employee(String name,String role){

this.name = name;
this.role= role;
}

publicLonggetId(){
returnthis.id;
}

publicStringgetName(){
returnthis.name;
}

publicStringgetRole(){
returnthis.role;
}

publicvoidsetId(Long id){
this.id = id;
}

publicvoidsetName(String name){
this.name = name;
}

publicvoidsetRole(String role){
this.role= role;
}

@Override
publicbooleanequals(Object o){

if(this== o)
returntrue;
if(!(o instanceofEmployee))
returnfalse;
Employeeemployee=(Employee) o;
returnObjects.equals(this.id, employee.id)&&Objects.equals(this.name, employee.name)
&&Objects.equals(this.role,employee.role);
}
@Override
publicinthashCode(){
returnObjects.hash(this.id,this.name,this.role);
}

@Override
publicStringtoString(){
return"Employee{"+"id="+this.id +", name='"+this.name +'\''+", role='"+this.role+'\''+'}';
}
}
```

Despite being small, this Java class contains much:

- @Entity is a JPA annotation to make this object ready for storage in a JPA-based data store.
- id, name, and role are attributes of our Employee domain object. id is marked with more JPA annotations to indicate it’s the primary key and automatically populated by the JPA provider.
- a custom constructor is created when we need to create a new instance, but don’t yet have an id.

With this domain object definition, we can now turn to Spring Data JPA to handle the tedious database interactions.

Spring Data JPA repositories are interfaces with methods supporting creating, reading, updating, and deleting records against a back end data store. Some repositories also support data paging, and sorting, where appropriate. Spring Data synthesizes implementations based on conventions found in the naming of the methods in the interface.

There are multiple repository implementations besides JPA. You can use Spring Data MongoDB, Spring Data GemFire, Spring Data Cassandra, etc. For this tutorial, we’ll stick with JPA.

Spring makes accessing data easy. By simply declaring the following EmployeeRepository interface we automatically will be able to

- Create new Employees
- Update existing ones
- Delete Employees
- Find Employees (one, all, or search by simple or complex properties)

nonrest/src/main/java/payroll/EmployeeRepository.java

```
package payroll;

importorg.springframework.data.jpa.repository.JpaRepository;

interfaceEmployeeRepositoryextendsJpaRepository<Employee,Long>{

}
```

To get all this free functionality, all we had to do was declare an interface which extends

Spring Data JPA's JpaRepository, specifying the domain type as Employee and the id type as Long.

Spring Data's repository solution makes it possible to sidestep data store specifics and instead solve a majority of problems using domain-specific terminology.

Believe it or not, this is enough to launch an application! A Spring Boot application is, at a minimum, a public static void main entry-point and the @SpringBootApplication annotation. This tells Spring Boot to help out, wherever possible.

nonrest/src/main/java/payroll/PayrollApplication.java

```
package payroll;

importorg.springframework.boot.SpringApplication;
importorg.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
publicclassPayrollApplication{

publicstaticvoidmain(String...args){
SpringApplication.run(PayrollApplication.class,args);
}
}
```

@SpringBootApplication is a meta-annotation that pulls in **component scanning**, **autoconfiguration**, and **property support**. We won't dive into the details of Spring Boot in this tutorial, but in essence, it will fire up a servlet container and serve up our service.

Nevertheless, an application with no data isn't very interesting, so let's preload it. The following class will get loaded automatically by Spring:

nonrest/src/main/java/payroll/LoadDatabase.java

```
package payroll;

importorg.slf4j.Logger;
importorg.slf4j.LoggerFactory;
importorg.springframework.boot.CommandLineRunner;
importorg.springframework.context.annotation.Bean;
importorg.springframework.context.annotation.Configuration;

@Configuration
classLoadDatabase{

privatestaticfinalLogger log =LoggerFactory.getLogger(LoadDatabase.class);

@Bean
CommandLineRunnerinitDatabase(EmployeeRepository repository){

returnargs->{
log.info("Preloading "+repository.save(newEmployee("Bilbo Baggins","burglar")));
log.info("Preloading "+repository.save(newEmployee("Frodo Baggins","thief")));
};
}
}
```

What happens when it gets loaded?

- Spring Boot will run ALL CommandLineRunner beans once the application context is loaded.
- This runner will request a copy of the EmployeeRepository you just created.
- Using it, it will create two entities and store them.

Right-click and **Run**PayRollApplication, and this is what you get:

Fragment of console output showing preloading of data

```
...
2018-08-09 11:36:26.169  INFO 74611 --- [main] payroll.LoadDatabase : Preloading Employee(id=1,
name=Bilbo Baggins, role=burglar)
2018-08-09 11:36:26.174  INFO 74611 --- [main] payroll.LoadDatabase : Preloading Employee(id=2,
name=Frodo Baggins, role=thief)
...
```

This isn't the **whole** log, but just the key bits of preloading data. (Indeed, check out the whole console. It's glorious.)

HTTP is the Platform

To wrap your repository with a web layer, you must turn to Spring MVC. Thanks to Spring Boot, there is little in infrastructure to code. Instead, we can focus on actions:

nonrest/src/main/java/payroll/EmployeeController.java

```
package payroll;

importjava.util.List;

importorg.springframework.web.bind.annotation.DeleteMapping;
importorg.springframework.web.bind.annotation.GetMapping;
importorg.springframework.web.bind.annotation.PathVariable;
importorg.springframework.web.bind.annotation.PostMapping;
importorg.springframework.web.bind.annotation.PutMapping;
importorg.springframework.web.bind.annotation.RequestBody;
importorg.springframework.web.bind.annotation.RestController;

@RestController
classEmployeeController{

privatefinalEmployeeRepository repository;

EmployeeController(EmployeeRepository repository){
this.repository= repository;
}


// Aggregate root
// tag::get-aggregate-root[]
@GetMapping("/employees")
List<Employee>all(){
returnrepository.findAll();
}
// end::get-aggregate-root[]

@PostMapping("/employees")
EmployeenewEmployee(@RequestBodyEmployeenewEmployee){
returnrepository.save(newEmployee);
}

// Single item

@GetMapping("/employees/{id}")
Employeeone(@PathVariableLong id){

returnrepository.findById(id)
.orElseThrow(()->newEmployeeNotFoundException(id));
}

@PutMapping("/employees/{id}")
EmployeereplaceEmployee(@RequestBodyEmployeenewEmployee,@PathVariableLong id){

returnrepository.findById(id)
.map(employee ->{
employee.setName(newEmployee.getName());
employee.setRole(newEmployee.getRole());
returnrepository.save(employee);
})
.orElseGet(()->{
newEmployee.setId(id);
returnrepository.save(newEmployee);
});
}

@DeleteMapping("/employees/{id}")
voiddeleteEmployee(@PathVariableLong id){
repository.deleteById(id);
}
}
```

- @RestController indicates that the data returned by each method will be written straight into the response body instead of rendering a template.
- An EmployeeRepository is injected by constructor into the controller.
- We have routes for each operation (@GetMapping, @PostMapping, @PutMapping and @DeleteMapping, corresponding to HTTP GET, POST, PUT, and DELETE calls). (NOTE: It's useful to read each method and understand what they do.)
- EmployeeNotFoundException is an exception used to indicate when an employee is looked up but not found.

nonrest/src/main/java/payroll/EmployeeNotFoundException.java

```
package payroll;

classEmployeeNotFoundExceptionextendsRuntimeException{

EmployeeNotFoundException(Long id){
super("Could not find employee "+ id);
}
}
```

When an EmployeeNotFoundException is thrown, this extra tidbit of Spring MVC configuration is used to render an **HTTP 404**:

nonrest/src/main/java/payroll/EmployeeNotFoundAdvice.java

```
package payroll;

importorg.springframework.http.HttpStatus;
importorg.springframework.web.bind.annotation.ControllerAdvice;
importorg.springframework.web.bind.annotation.ExceptionHandler;
importorg.springframework.web.bind.annotation.ResponseBody;
importorg.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
classEmployeeNotFoundAdvice{

@ResponseBody
@ExceptionHandler(EmployeeNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
StringemployeeNotFoundHandler(EmployeeNotFoundException ex){
returnex.getMessage();
}
}
```

- @ResponseBody signals that this advice is rendered straight into the response body.
- @ExceptionHandler configures the advice to only respond if an EmployeeNotFoundException is thrown.
- @ResponseStatus says to issue an HttpStatus.NOT_FOUND, i.e. an **HTTP 404**.
- The body of the advice generates the content. In this case, it gives the message of the exception.

To launch the application, either right-click the public static void main in PayRollApplication and select **Run** from your IDE, or:

Spring Initializr uses maven wrapper so type this:

```
$ ./mvnw clean spring-boot:run
```

Alternatively using your installed maven version type this:

```
$ mvn clean spring-boot:run
```

When the app starts, we can immediately interrogate it.

```
$ curl -v localhost:8080/employees
```

This will yield:

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /employees HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Thu, 09 Aug 2018 17:58:00 GMT
<
* Connection #0 to host localhost left intact
[{"id":1,"name":"Bilbo Baggins","role":"burglar"},{"id":2,"name":"Frodo Baggins","role":"thief"}]
```

Here you can see the pre-loaded data, in a compacted format.

If you try and query a user that doesn't exist…

```
$ curl -v localhost:8080/employees/99
```

You get…

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /employees/99 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 404
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 26
< Date: Thu, 09 Aug 2018 18:00:56 GMT
<
* Connection #0 to host localhost left intact
Could not find employee 99
```

This message nicely shows an **HTTP 404** error with the custom message **Could not find employee 99**.

It's not hard to show the currently coded interactions…

If you are using Windows Command Prompt to issue cURL commands, chances are the below command won't work properly. You must either pick a terminal that support single quoted arguments, or use double quotes and then escape the ones inside the JSON.

To create a new Employee record we use the following command in a terminal—the $ at the beginning signifies that what follows it is a terminal command:

```
$ curl -X POST localhost:8080/employees -H 'Content-type:application/json' -d '{"name": "Samwise Gamgee",
"role": "gardener"}'
```

Then it stores newly created employee and sends it back to us:

```
{"id":3,"name":"Samwise Gamgee","role":"gardener"}
```

You can update the user. Let’s change his role.

```
$ curl -X PUT localhost:8080/employees/3 -H 'Content-type:application/json' -d '{"name": "Samwise Gamgee", "role": "ring bearer"}'
```

And we can see the change reflected in the output.

```
{"id":3,"name":"Samwise Gamgee","role":"ring bearer"}
```

The way you construct your service can have significant impacts. In this situation, we said **update**, but **replace** is a better description. For example, if the name was NOT provided, it would instead get nulled out.

Finally, you can delete users like this:

```
$ curl -X DELETE localhost:8080/employees/3

# Now if we look again, it's gone
$ curl localhost:8080/employees/3
Could not find employee 3
```

This is all well and good, but do we have a RESTful service yet? (If you didn’t catch the hint, the answer is no.)

What’s missing?

What makes something RESTful?

So far, you have a web-based service that handles the core operations involving employee data. But that’s not enough to make things "RESTful".

- Pretty URLs like /employees/3 aren’t REST.
- Merely using GET, POST, etc. isn’t REST.
- Having all the CRUD operations laid out isn’t REST.

In fact, what we have built so far is better described as **RPC** (**Remote Procedure Call**). That’s because there is no way to know how to interact with this service. If you published this today, you’d also have to write a document or host a developer’s portal somewhere with all the details.

This statement of Roy Fielding’s may further lend a clue to the difference between **REST** and **RPC**:

I am getting frustrated by the number of people calling any HTTP-based interface a REST API. Today’s example is the SocialSite REST API. That is RPC. It screams RPC. There is so much coupling on display that it should be given an X rating.

What needs to be done to make the REST architectural style clear on the notion that hypertext is a constraint? In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API. Period. Is there some broken manual somewhere that needs to be fixed?

— Roy Fielding
https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

The side effect of NOT including hypermedia in our representations is that clients MUST hard code URIs to navigate the API. This leads to the same brittle nature that predated the rise of e-commerce on the web. It's a signal that our JSON output needs a little help.

Introducing Spring HATEOAS, a Spring project aimed at helping you write hypermedia-driven outputs. To upgrade your service to being RESTful, add this to your build:

Adding Spring HATEOAS to dependencies section of pom.xml

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

This tiny library will give us the constructs to define a RESTful service and then render it in an acceptable format for client consumption.

A critical ingredient to any RESTful service is adding links to relevant operations. To make your controller more RESTful, add links like this:

Getting a single item resource

```
@GetMapping("/employees/{id}")
EntityModel<Employee>one(@PathVariableLong id){

Employeeemployee=repository.findById(id)//
.orElseThrow(()->newEmployeeNotFoundException(id));

returnEntityModel.of(employee,//
    linkTo(methodOn(EmployeeController.class).one(id)).withSelfRel(),
    linkTo(methodOn(EmployeeController.class).all()).withRel("employees"));
}
```

This tutorial is based on Spring MVC and uses the static helper methods from WebMvcLinkBuilder to build these links. If you are using Spring WebFlux in your project, you must instead use WebFluxLinkBuilder.

This is very similar to what we had before, but a few things have changed:

- The return type of the method has changed from Employee to EntityModel<Employee>. EntityModel<T> is a generic container from Spring HATEOAS that includes not only the data but a collection of links.
- linkTo(methodOn(EmployeeController.class).one(id)).withSelfRel() asks that Spring HATEOAS build a link to the EmployeeController 's one() method, and flag it as a self link.
- linkTo(methodOn(EmployeeController.class).all()).withRel("employees") asks Spring HATEOAS to build a link to the aggregate root, all(), and call it "employees".

What do we mean by "build a link"? One of Spring HATEOAS's core types is Link. It includes a **URI** and a **rel** (relation). Links are what empower the web. Before the World Wide

Web, other document systems would render information or links, but it was the linking of documents WITH this kind of relationship metadata that stitched the web together.

Roy Fielding encourages building APIs with the same techniques that made the web successful, and links are one of them.

If you restart the application and query the employee record of *Bilbo*, you'll get a slightly different response than earlier:

Curling prettier

When your curl output gets more complex it can become hard to read. Use this or other tips to prettify the json returned by curl:

```
# The indicated part pipes the output to json_pp and asks it to make your JSON pretty. (Or use whatever tool you like!)
#                                  v------------------v
curl -v localhost:8080/employees/1 | json_pp
```

RESTful representation of a single employee

```
{
"id":1,
"name":"Bilbo Baggins",
"role":"burglar",
"_links":{
"self":{
"href":"http://localhost:8080/employees/1"
},
"employees":{
"href":"http://localhost:8080/employees"
}
}
}
```

This decompressed output shows not only the data elements you saw earlier (id, name and role), but also a _links entry containing two URIs. This entire document is formatted using HAL.

HAL is a lightweight mediatype that allows encoding not just data but also hypermedia controls, alerting consumers to other parts of the API they can navigate toward. In this case, there is a "self" link (kind of like a this statement in code) along with a link back to the **aggregate root**.

To make the aggregate root ALSO more RESTful, you want to include top level links while ALSO including any RESTful components within.

So we turn this

Getting an aggregate root

```
@GetMapping("/employees")
List<Employee>all(){
returnrepository.findAll();
}
```

into this

Getting an aggregate root **resource**

```
@GetMapping("/employees")
CollectionModel<EntityModel<Employee>>all(){

List<EntityModel<Employee>> employees =repository.findAll().stream()
.map(employee ->EntityModel.of(employee,
        linkTo(methodOn(EmployeeController.class).one(employee.getId())).withSelfRel(),
        linkTo(methodOn(EmployeeController.class).all()).withRel("employees")))
.collect(Collectors.toList());

returnCollectionModel.of(employees,linkTo(methodOn(EmployeeController.class).all()).withSelfRel());
}
```

Wow! That method, which used to just be repository.findAll(), is all grown up! Not to worry. Let's unpack it.

CollectionModel<> is another Spring HATEOAS container; it's aimed at en

Building an application usingMaven

Maven is one of the open-source Java build tools developed by Apache Software Foundation. It can compile, test, and package a java program into .jar or .war format.

Maven makes use of the pom.xml file to build java projects.

Project Object Model (POM) is an XML file that contains the java project details, configurations, and settings required for maven to build the project.

The **pom.xml** file is present in the root of the java project directory. Primarily it contains the project dependencies.

For example, when a developer wants to implement a PostgreSQL database connectivity functionality, he will make use of the PostgreSQL JDBC Driver dependency from the maven repository by adding it to the pom.xml file.

So when you build the code with maven, it reads the pom.xml file and downloads all the dependencies from the maven repository. Dependencies could be third-party libraries from the public Maven Repository or common libraries hosted within an organization's private maven repository. You can compare it with Python pip, Nodejs npm, or Ruby gems

Commonly organizations use Sonatyope nexus as a private hosted maven repository.

By default, maven uses the public repository but if you have in-house private maven repositories, you configure custom maven repository URLs in settings.xml maven configuration present in the maven installation directory. for example, /opt/apache-maven-3.8.6/conf/settings.xml

Maven Prerequisites

For maven to work you need the following installed on your system

1. Java JDK
2. Maven

To install and configure JDK and maven, follow the **maven installation guide.**

Build Java Application Using Maven

For this example, we will be using the open-source **java spring boot application** named pet-clinic.

First, clone the application to your development machine or server.

```
git clone https://github.com/spring-projects/spring-petclinic.git
```

The code base has the following important folders and files. It is common in real-time project code as well.

1. **/src folder:** This folder contains the source code based on the java spring framework.
2. **/src/tests folder:** This folder contains the unit tests & integration tests of the code under the tests folder.
3. **pom.xml file:** It contains all the dependencies required for the pet-clinic applications. As it is an open-source application, all the dependencies are from the public maven repository.

To build the project, cd into the project root directory. In my case its spring-petclinic. It should contain the pom.xml file

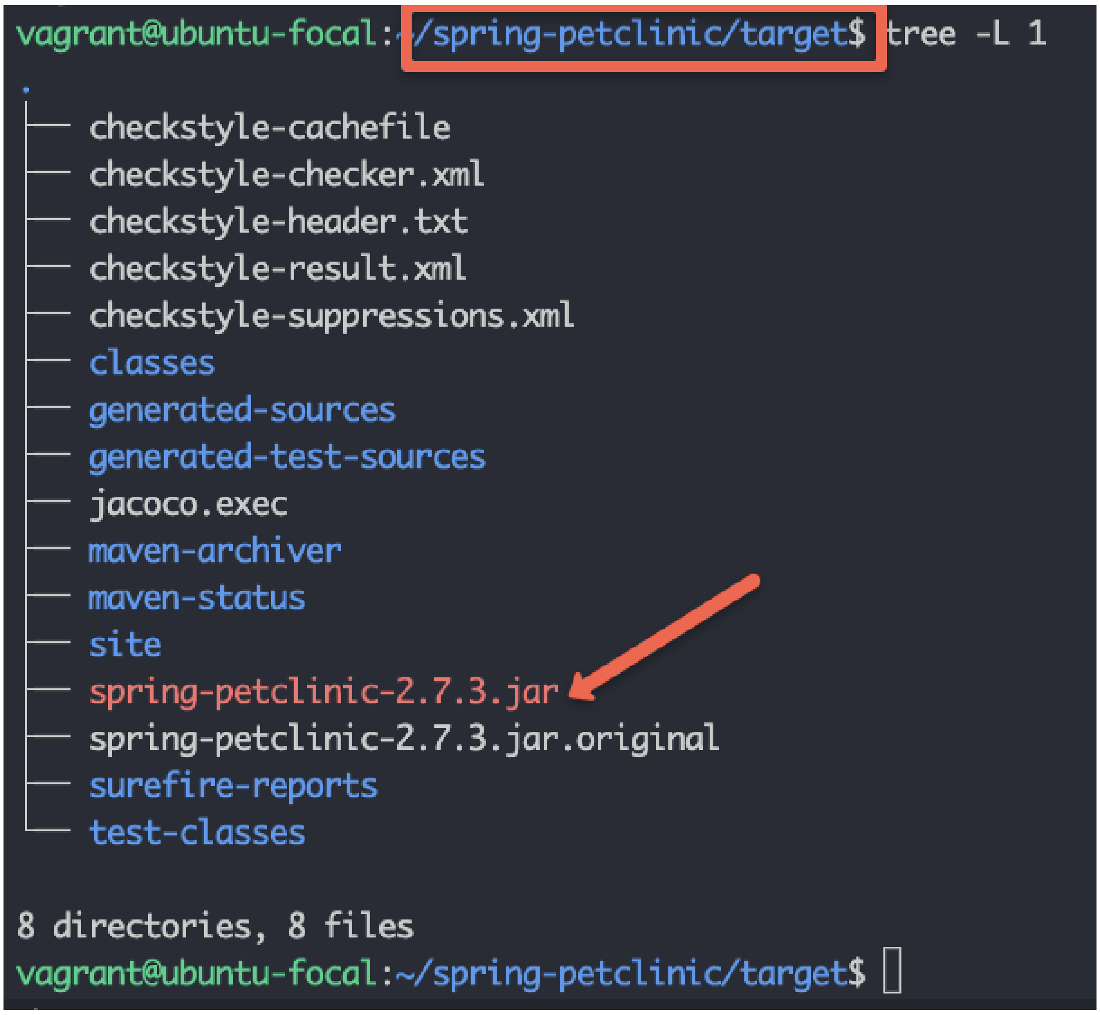
```
cd spring-petclinic
```

From a CI perspective, we just have to **build, test, and package** the project to create a deployable artifact(jar file)

So commonly in the CI process, we build and package the java projects using the following maven command. It compiles the code, tests it, package it as a jar file in the target folder, and will also install(copy) the jar package in the local .m2 repository.

```
mvn clean install
```

After executing the above command, you will see a folder named **target** in the root directory. Inside the target directory, you will see the packaged jar file as shown below. We call it a deployable artifact.



```
vagrant@ubuntu-focal:~/spring-petclinic/target$ tree -L 1
.
├── checkstyle-cachefile
├── checkstyle-checker.xml
├── checkstyle-header.txt
├── checkstyle-result.xml
├── checkstyle-suppressions.xml
├── classes
├── generated-sources
├── generated-test-sources
├── jacoco.exec
├── maven-archiver
├── maven-status
├── site
├── spring-petclinic-2.7.3.jar
├── spring-petclinic-2.7.3.jar.original
├── surefire-reports
└── test-classes

8 directories, 8 files
vagrant@ubuntu-focal:~/spring-petclinic/target$
```

Even time you run **mvn clean install**, it deletes target directory and packages from the local **.m2** repository and replaces it with the latest build files and packages.

If you want to skip the test during build, you can add the **-Dmaven.test.skip=true** parameter as shown below.

```
mvn clean install -Dmaven.test.skip=true
```

Now that you have understood how to build a java project using maven, let’s look into the maven lifecycle. Few commands we don’t have to use in the CI pipelines. However, it is good to know about the maven lifecycle commands and you can use them depending on your CI pipeline requirement.

Maven Lifecycle Explained

Let’s take a look at each maven lifecycle phase in order. Each phase executes all the phases before it. For example, if you execute the third phase, one, two, and three get executed.

1. Maven Validate (mvn validate)

mvn validate validates the maven project. It downloads all the required dependencies to the local **.m2** repository.

2. Maven Compile (mvn compile)

mvn compile compiles the java project. It runs validate first and then compiles the code.

3. Maven Test (mvn test)

mvn test command runs the unit test that is part of the code. You can test classes individually, methods individually, or add patterns to run tests on all methods that match the pattern.

4. Maven Package (mvn package)

mvn package commands compile the code, test it and finally package it in the required format (jar or war)

5. Maven Verify (mvn verify)

mvn verify command runs all the phases explained before in order and runs checks on integration tests and checkstyles if they are defined in the project.

6. Maven Install (mvn install)

mvn install command installs the packaged code in the local maven repository.

7. Maven Deploy (mvn deploy)

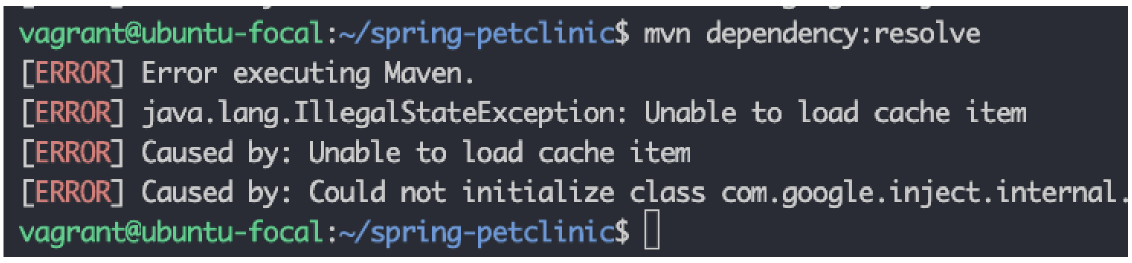
mvn deploy command, deploys the package to the remote maven repository. When you run deploy, it first runs validate, compile, test, package, verify, install, and then finally deploys the package to the remote maven repository.

Possible Maven Build Errors

java.lang.IllegalStateException: Unable to load cache item

If maven doesn't support the Java version, you will get the above error.

To rectify it, install the latest maven version that supports the installed Java version.



```
vagrant@ubuntu-focal:~/spring-petclinic$ mvn dependency:resolve
[ERROR] Error executing Maven.
[ERROR] java.lang.IllegalStateException: Unable to load cache item
[ERROR] Caused by: Unable to load cache item
[ERROR] Caused by: Could not initialize class com.google.inject.internal.
vagrant@ubuntu-focal:~/spring-petclinic$
```

If you try to execute the maven command from the location where there is no **pom.xml** file, you will get the following error.

```
The goal you specified requires a project to execute but there is no POM in this directory
```

To rectify this, execute the maven command from the folder that has the pom.xml file.

Maven Build FAQs

Does mvn package run tests?

Yes. By default, the `mvn package` command runs the test. However, you can add the flag `-Dmaven.test.skip` to skip the tests.

What does Maven test do?

mvn test runs all the unit tests for the java project.

Conclusion

As a Devops engineer, it is very important to understand the java build process if you are working on deploying java projects.