**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.

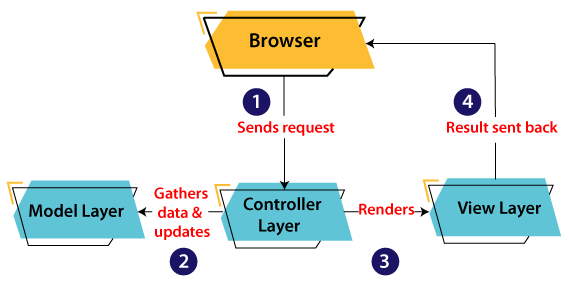
What is MVC architecture in Java?

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.

**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.getDepartment());

       }

    }

**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**

**public** **class** MVCMain {

**public** **static** **void** main(String[] args) {

          Employee model = retriveEmployeeFromDatabase();

          EmployeeView view = **new** EmployeeView();

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

          controller.updateView();

          controller.setEmployeeName("Hitesh");

          System.out.println("\n Employee Details after updating: ");

          controller.updateView();

       }

**private** **static** Employee retriveEmployeeFromDatabase(){

          Employee Employee = **new** Employee();

          Employee.setName("Amit");

          Employee.setId("11");

          Employee.setDepartment("UPES");

**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: Amit

Employee ID: 11

Employee Department: UPES

Employee Details after updating:

Name: Hitesh

Employee ID: 11

Employee Department: UPES

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