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.

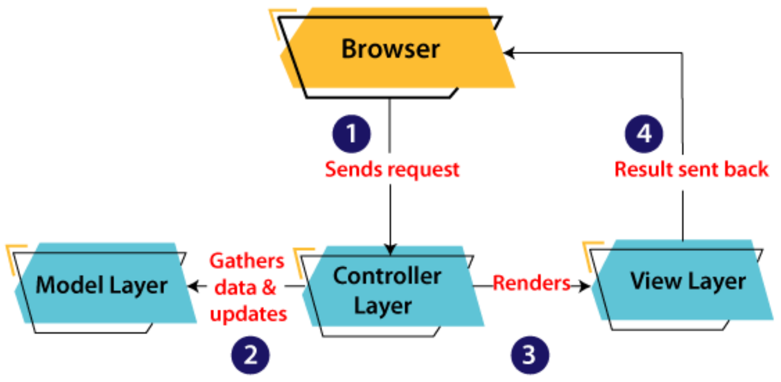
What is MVC architecture in Java?

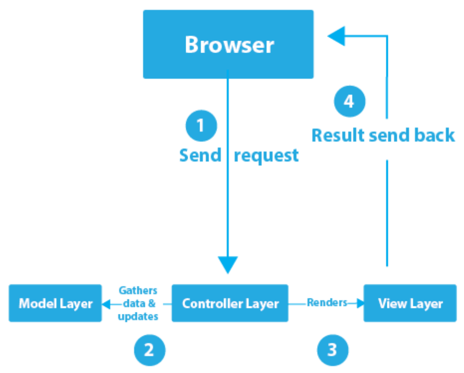
The model designs based on the MVC architecture follow MVC design pattern. The application logic is separated from the user interface while designing the software using model designs.

The MVC pattern architecture consists of three layers:

1. **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.
2. **View:** It represents the presentation layer of application. It is used to visualize the data that the model contains.
3. **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:





MVC Architecture in Java

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.
- **Separation of Concerns:** MVC promotes a clear separation of concerns between the model, view, and controller components. This separation allows for better code organization, improved modularity, and easier maintenance. Developers can focus on specific aspects of the application without impacting other components.

- **Code Reusability:** By separating the concerns into distinct components, code reuse becomes more feasible. The model can be reused across different views, and multiple views can be created for a single model. This reusability reduces duplication of code and improves development efficiency.
- **Simultaneous Development:** MVC allows multiple developers to work simultaneously on different components. The model, view, and controller can be developed independently as long as they adhere to the defined interfaces and communication protocols. This parallel development approach accelerates the overall development process.
- **Flexibility and Extensibility:** MVC provides flexibility by allowing changes in one component without affecting others. For example, modifying the view does not require altering the model or controller. This flexibility also enables the easy addition of new views or controllers to enhance the application's functionality.
- **Testability:** The separation of concerns in MVC makes unit testing and debugging more manageable. Each component can be independently tested, as they have well-defined responsibilities and interfaces. This promotes comprehensive testing, reduces dependencies, and improves the overall quality of the application.
- **Enhanced User Experience:** With MVC, the view layer handles the presentation of data to the user. This separation allows for greater control over the user interface and enables the use of different views for different platforms or devices. Developers can create responsive and user-friendly interfaces tailored to specific user needs.
- **Support for Maintainability:** MVC simplifies the maintenance of applications over time. Changes can be made to individual components without requiring extensive modifications to the entire system. This modularity enhances maintainability and reduces the risk of introducing bugs or breaking existing functionality.

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

```
// 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.

Fuente:

https://www.javatpoint.com/mvc-architecture-in-java

https://www.prepbytes.com/blog/java/mvc-architecture-in-java/#:~:text=In%20the%20MVC%20architecture%2C%20the,Model%20layer%20for%20data%20processing.