Model View Controller MVC

About us

Pablo Medina Moreno

alu0101481449@ull.edu.es

Sergio Pérez Lozano

alu0101473260@ull.edu.es

Table of contents

Brief History

Trygve Reenskaug

- Formulated the model–view–controller (MVC) in 1979
- In the Xerox Palo Alto Research Center (PARC).

Introduction

What's the Model-View-Controller (MVC) ?

“It is a used architectural pattern in software development to separate business logic from user interface and data management”

Introduction

Its main purpose is to achieve a clear separation of responsibilities.

The MVC pattern has three components:

1. Model
2. View
3. Controller

Model

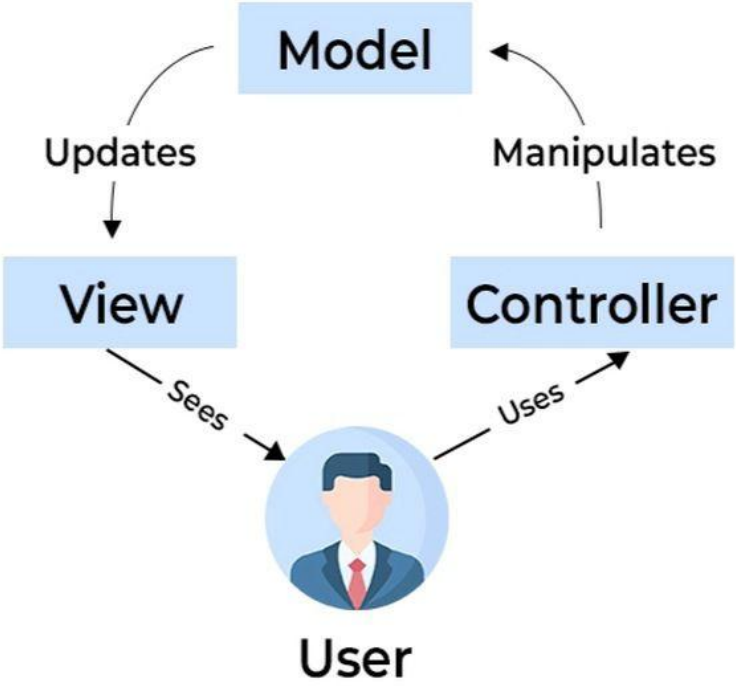
- Manages the application's data and provides methods to access it
- Encapsulates application state
- Doesn't know anything about the view

View

- The application's user interface.
- It is responsible for displaying data to the user and capturing user interactions.
- No logic, just it simply presents the information in a suitable manner for the user to interact with.

Controller

- Intermediary between the model and the view
- Receives user interactions through the view
- Processes these interactions (requests to the model)
- Updates the view


Model
Updates
Manipulates
View
Controller
Sees
Uses
User

Analogy with a restaurant

Model

View

Controller

Brief Example

Software Application to show and update top football scorers.

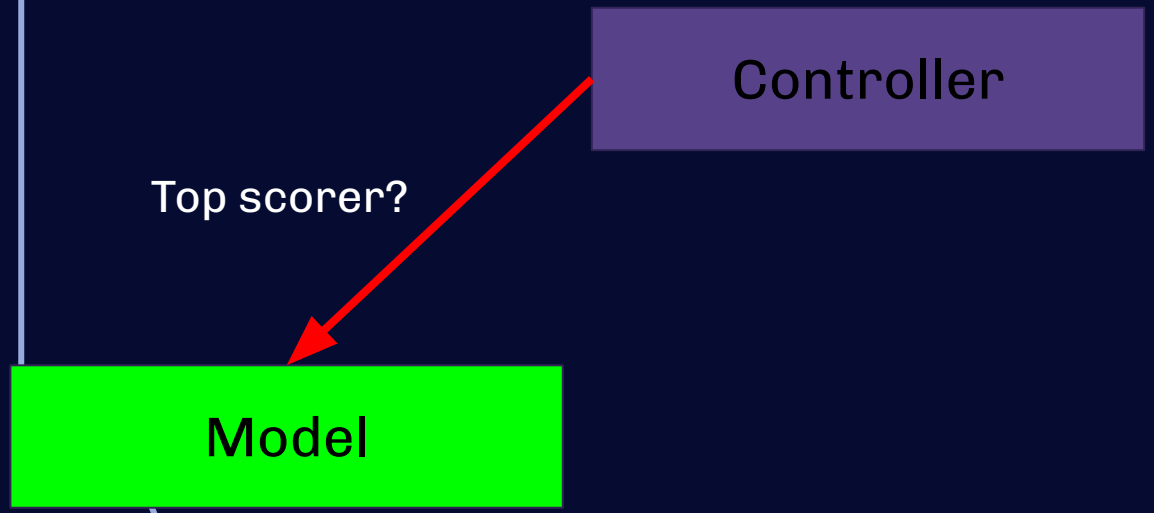
Brief Example

Database → Goals per Player

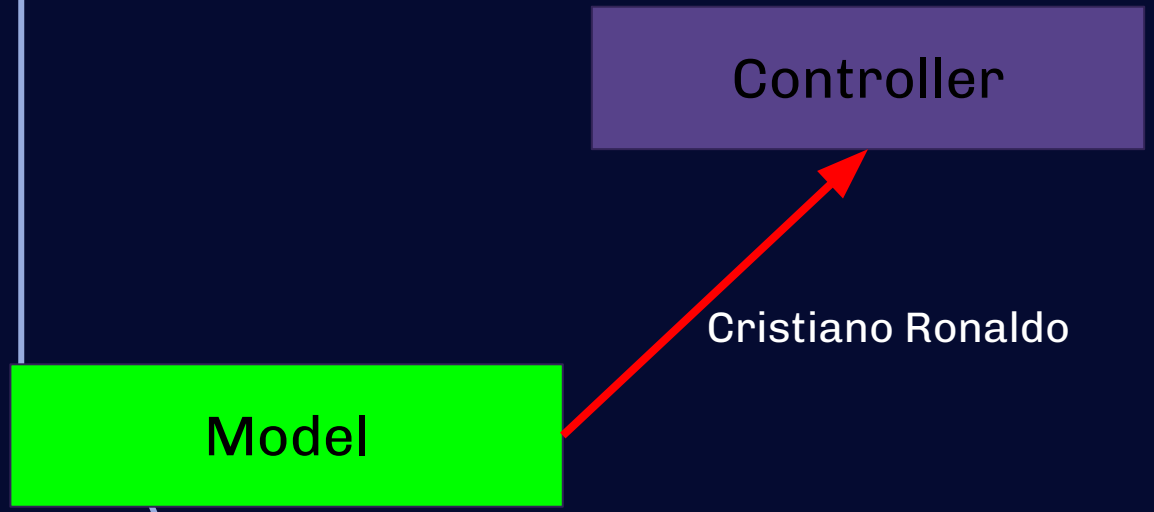
Player	Goals
Leo Messi	826
Cristiano Ronaldo	882
Suso Santana	288
Sergio Pérez	0

Brief Example

Brief Example



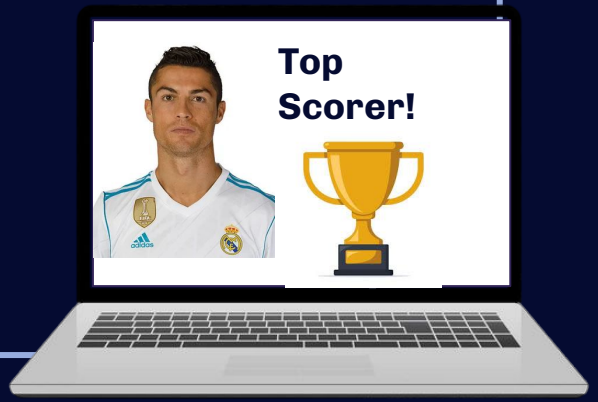
Brief Example



Brief Example

Controller

View



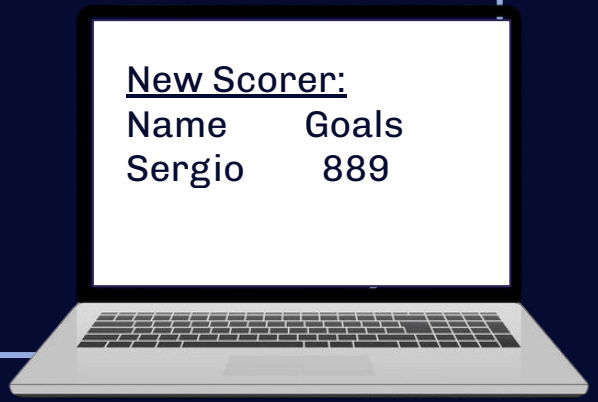
Model

What would happen if a new event were to occur now?

Brief Example

Controller

View

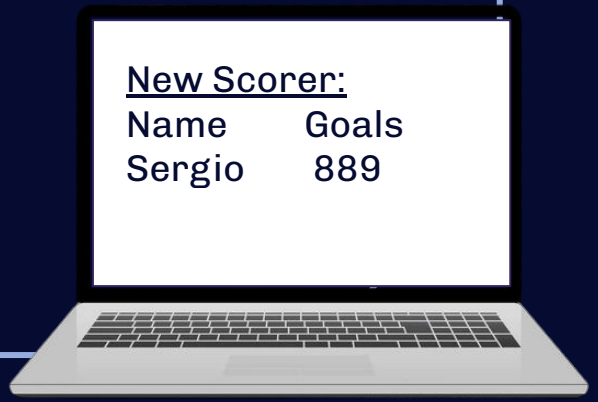


Model

Brief Example

Controller

View



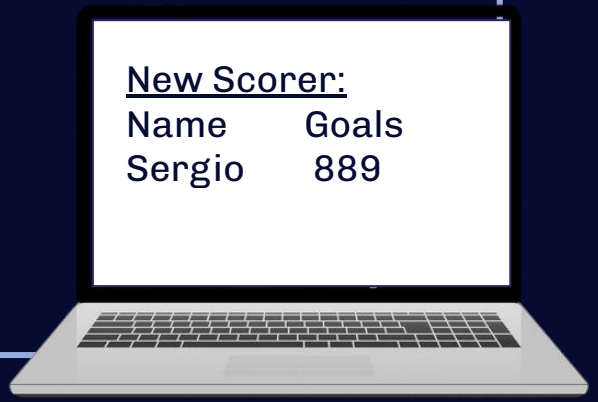
Model

Brief Example

Controller

Update();
Top Scorer;

View

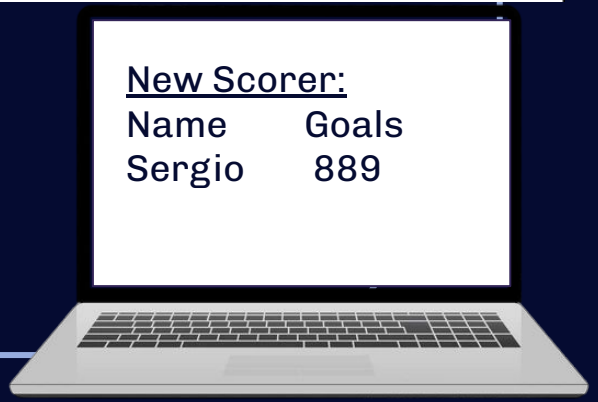


Model

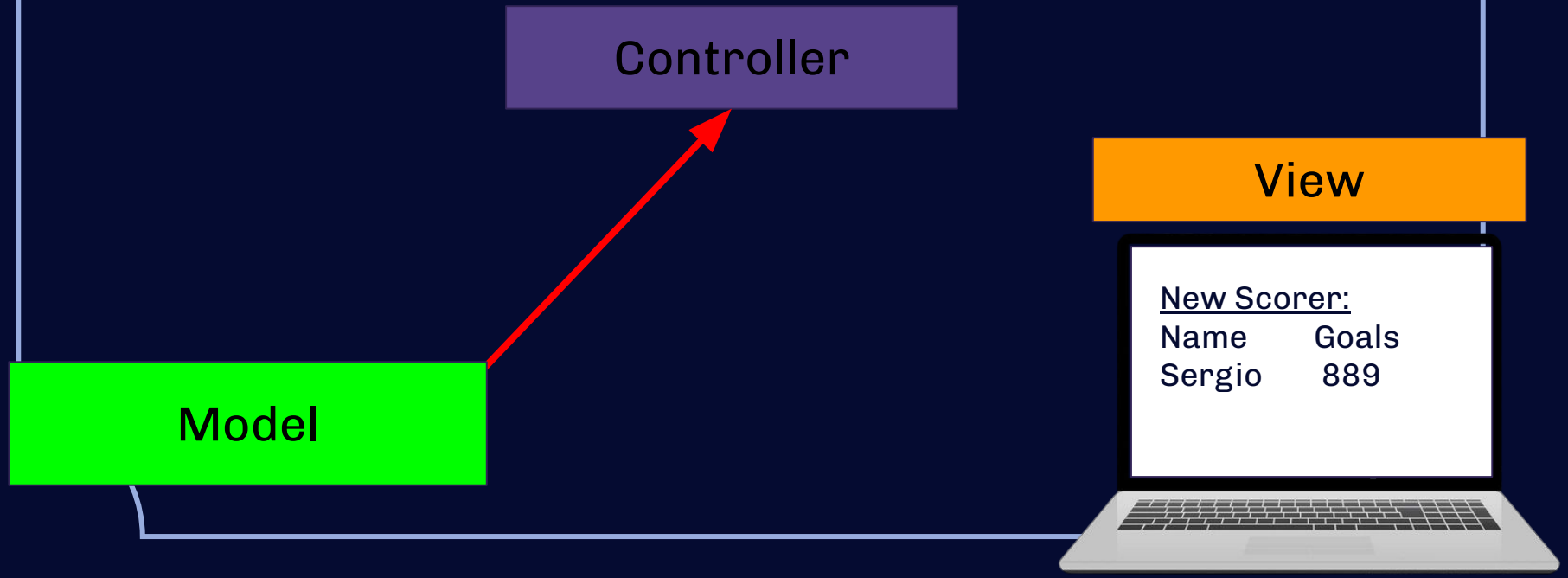
Brief Example

Player	Goals
Leo Messi	826
Cristiano Ronaldo	882
Suso Santana	288
Sergio Pérez	889

Model



Brief Example



Brief Example

Controller

View

Model

Example

User	Grade
Ricardo	4.8

Basic Structure

```
class Model {
  constructor() {}
}

class View {
  constructor() {}
}

class Controller {
  constructor(private model: Model, private view: View) {
    this.model = model
    this.view = view
  }
}

const app = new Controller(new Model(), new View())
```

Main

```
function main() {
  const userModel = new UserModel("Ricardo", 4.8);
  const userView = new UserView();
  const userController = new UserController(userModel, userView);

  // Updating the user's data through the controller
  userController.updateName("Pedro");
  userController.updateGrade(5);
}

main();
```

Bad practice!

```
//Controller
export class UserController {
  constructor(private userModel: UserModel, private view: UserView) {
      this.userModel = userModel;
      this.view = view;
  }
  public updateName(name: string): void {
      this.userModel.name = name;
      this.updateView();
  }
  public updateGrade(grade: number): void {
      this.userModel.grade = grade;
      this.updateView();
  }
  public updateView(): void {
      console.log(`Name: ${this.userModel.name}, Grade: ${this.userModel.grade}`);
  }
}
```

```
//Model
export class UserModel {
  public name: string;
  public grade: number;

  constructor(name: string, grade: number) {
      this.name = name;
      this.grade = grade;
  }
}
```

Good practice!

```typescript
import { UserModel } from './model.js';
import { UserView } from './view.js';

//Controller
export class UserController {
  private userModel: UserModel;
  private view: UserView;

  constructor(userModel: UserModel, view: UserView) {
    this.userModel = userModel;
    this.view = view;
  }

  public updateName(name: string): void {
    this.userModel.setName(name);
    this.updateView();
  }

  public updateGrade(grade: number): void {
    this.userModel.setGrade(grade);
    this.updateView();
  }

  private updateView(): void {
    this.view.showUser(this.userModel);
  }
}
```

```typescript
//Model
export class UserModel {
  private name: string;
  private grade: number;

  constructor(name: string, grade: number) {
    this.name = name;
    this.grade = grade;
  }
  getName(): string {
    return this.name;
  }
  setName(name: string): void {
    this.name = name;
  }
  getGrade(): number {
    return this.grade;
  }
  setGrade(grade: number): void {
    this.grade = grade;
  }
}
```

Bad practice!

```
//Controller
export class UserController {
  constructor(private userModel: UserModel, private view: UserView) {
      this.userModel = userModel;
      this.view = view;
  }
  public updateName(name: string): void {
      this.userModel.name = name;
      this.updateView();
  }
  public updateGrade(grade: number): void {
      this.userModel.grade = grade;
      this.updateView();
  }
  public updateView(): void {
      console.log(`Name: ${this.userModel.name}, Grade: ${this.userModel.grade}`);
  }
}
```

Good practice!

```typescript
import { UserModel } from './model.js';
import { UserView } from './view.js';

//Controller
export class UserController {
  private userModel: UserModel;
  private view: UserView;

  constructor(userModel: UserModel, view: UserView) {
    this.userModel = userModel;
    this.view = view;
  }

  public updateName(name: string): void {
    this.userModel.setName(name);
    this.updateView();
  }

  public updateGrade(grade: number): void {
    this.userModel.setGrade(grade);
    this.updateView();
  }

  private updateView(): void {
    this.view.showUser(this.userModel);
  }
}
```

```typescript
export class UserView {

  /**
   * Constructor of the UserView class
   */
  constructor() {}

  /**
   * Method that shows the user's data
   * @param userModel is an instance of the User class
   */
  public showUser(userModel: UserModel): void {
    console.log(`Name: ${userModel.getName()}, Grade: ${userModel.getGrade()}`);
  }
}
```

Example: Todo List

The way to do it

- Model → Manages the data of an application
- View → A visual representation of the model
- Controller → Links the user and the system

```
class Model {
  constructor() {}
}

class View {
  constructor() {}
}

class Controller {
  constructor(private model: Model, private view: View) {
    this.model = model
    this.view = view
  }
}

const app = new Controller(new Model(), new View())
```

The Model

- What data and business logic does my application need to function?
- How are the data structured and organized within the model?
- What methods or functions are necessary to manipulate the data effectively?

```
class Model {
  private todos: {id: number, text: string, complete: boolean}[];

  constructor() {
    this.todos = [
      {id: 1, text: 'Run a marathon', complete: false},
      {id: 2, text: 'Plant a garden', complete: false},
    ];
  }
```

```
11    public addTodo(todoText: string): void {
12      const todo = {
13        id: this.todos.length > 0 ? this.todos[this.todos.length - 1].id + 1 : 1,
14        text: todoText,
15        complete: false,
16      };
17
18      this.todos.push(todo);
19    }
20
21    public deleteTodo(id: number): void {
22      this.todos = this.todos.filter((todo) => todo.id !== id);
23    }
24
25    public toggleTodo(id: number): void {
26      this.todos = this.todos.map((todo) =>
27        todo.id === id ? {id: todo.id, text: todo.text, complete: !todo.complete} : todo,
28      );
29    }
30  }
```

The View

- How will the data be displayed?
- How should the user interact with my program?

Todos

Add todo

Submit

```typescript
class View {
  private app: HTMLElement;
  private title: HTMLHeadingElement;
  private form: HTMLFormElement;
  private input: HTMLInputElement;
  private submitButton: HTMLButtonElement;
  private todoList: HTMLUListElement;

  constructor() {
    //...
  }

  // Method to create an element with an optional CSS class
  private createElement(tag: string, className?: string): HTMLElement {
    const element = document.createElement(tag);

    if (className) element.classList.add(className);

    return element;
  }

  // Method to retrieve an element from the DOM
  private getElement(selector: string): HTMLElement | null {
    const element = document.querySelector(selector) as HTMLElement;
    return element;
  }

  // ...
}
```

```
9   constructor() {
10    // The root element
11    this.app = this.getElement('#root') as HTMLElement;
12
13    // The title of the application
14    this.title = this.createElement('h1') as HTMLHeadingElement;
15    this.title.textContent = 'Todos';
16
17    // The form, with a text input and a submit button
18    this.form = this.createElement('form') as HTMLFormElement;
19
20    this.input = this.createElement('input') as HTMLInputElement;
21    this.input.type = 'text';
22    this.input.placeholder = 'Add todo';
23    this.input.name = 'todo';
24
25    this.submitButton = this.createElement('button') as HTMLButtonElement;
26    this.submitButton.textContent = 'Submit';
27
28    // The visual representation of the todo list
29    this.todoList = this.createElement('ul', 'todo-list') as HTMLUListElement;
30
31    // Append the input and submit button to the form
32    this.form.append(this.input, this.submitButton);
33
34    // Append the title, form, and todo list to the app
35    this.app.append(this.title, this.form, this.todoList);
36  }
```

```
public displayTodos(todos: { id: number; text: string; complete: boolean }[]): void {
  // Display default message if there are no todos
  if (todos.length === 0) {
  const p = this.createElement('p') as HTMLParagraphElement;
  p.textContent = 'Nothing to do! Add a task?';
  this.todoList.append(p);
  } else {
```

```
    } else {
      // Create nodes for each todo
      todos.forEach(todo => {
      const li = this.createElement('li') as HTMLLIElement;
      li.id = todo.id.toString();

      const checkbox = this.createElement('input') as HTMLInputElement;
      checkbox.type = 'checkbox';
      checkbox.checked = todo.complete;

      const span = this.createElement('span') as HTMLSpanElement;
      span.contentEditable = 'true';
      span.classList.add('editable');

      if (todo.complete) {
        const strike = this.createElement('s') as HTMLElement;
        strike.textContent = todo.text;
        span.append(strike);
      } else {
        span.textContent = todo.text;
      }

      const deleteButton = this.createElement('button', 'delete') as HTMLButtonElement;
      deleteButton.textContent = 'Delete';
      li.append(checkbox, span, deleteButton);

      // Attach nodes to the todo list
      this.todoList.append(li);
    });
  }
}
```

The Controller

- What user events should be captured and processed by the controller?
- How do user actions relate to model operations and view updates?
- What methods or functions are necessary in the controller to efficiently coordinate between the model and the view?

```typescript
class Controller {
  /**
   * Model and View properties
   */
  private model: Model;
  private view: View;

  /**
   * Constructor of the Controller class
   * @param model is an instance of the Model class
   * @param view is an instance of the View class
   */
  constructor(model: Model, view: View) {
    this.model = model;
    this.view = view;

    // Show initial todos
    this.onTodoListChanged(this.model.getTodos());
  }

  /**
   * Method that displays the todos
   * @param todos is an array of todos
   */
  private onTodoListChanged = (todos: { id: number; text: string; complete: boolean }[]) : void => {
    this.view.displayTodos(todos);
  }
```

```typescript
public onTodoListChanged = (todos: { id: number; text: string; complete: boolean }[]) => {
  this.view.displayTodos(todos);
}

public handleAddTodo = (todoText: string) : void => {
  this.model.addTodo(todoText);
}

public handleDeleteTodo = (id: number) : void => {
  this.model.deleteTodo(id);
}

public handleToggleTodo = (id: number) : void => {
  this.model.toggleTodo(id);
}
```

Setting up event listeners

Now we have these handlers, but the controller still doesn't know when to call them

→ We have to put event listeners on the DOM elements in the view

View

```
public bindAddTodo(handler: (todoText: string) => void): void {
  this.form.addEventListener('submit', event => {
    event.preventDefault();

    if (this.todoText) {
      handler(this.todoText);
      this.resetInput();
    }
  });
}

public bindDeleteTodo(handler: (id: number) => void): void {
  this.todoList.addEventListener('click', event => {
    if ((event.target as HTMLElement).className === 'delete') {
      const id = parseInt((event.target as HTMLElement).parentElement!.id);
      handler(id);
    }
  });
}

public bindToggleTodo(handler: (id: number) => void): void {
  this.todoList.addEventListener('change', event => {
    if ((event.target as HTMLInputElement).type === 'checkbox') {
      const id = parseInt((event.target as HTMLElement).parentElement!.id);
      handler(id);
    }
  });
```

Controller

```
constructor(model: Model, view: View) {
  this.model = model;
  this.view = view;

  this.view.bindAddTodo(this.handleAddTodo);
  this.view.bindDeleteTodo(this.handleDeleteTodo);
  this.view.bindToggleTodo(this.handleToggleTodo);

  // Show initial todos
  this.onTodoListChanged(this.model.getTodos());
}
```

Now when a submit, click or change event happens on the specified elements, the corresponding handlers will be invoked.

Model

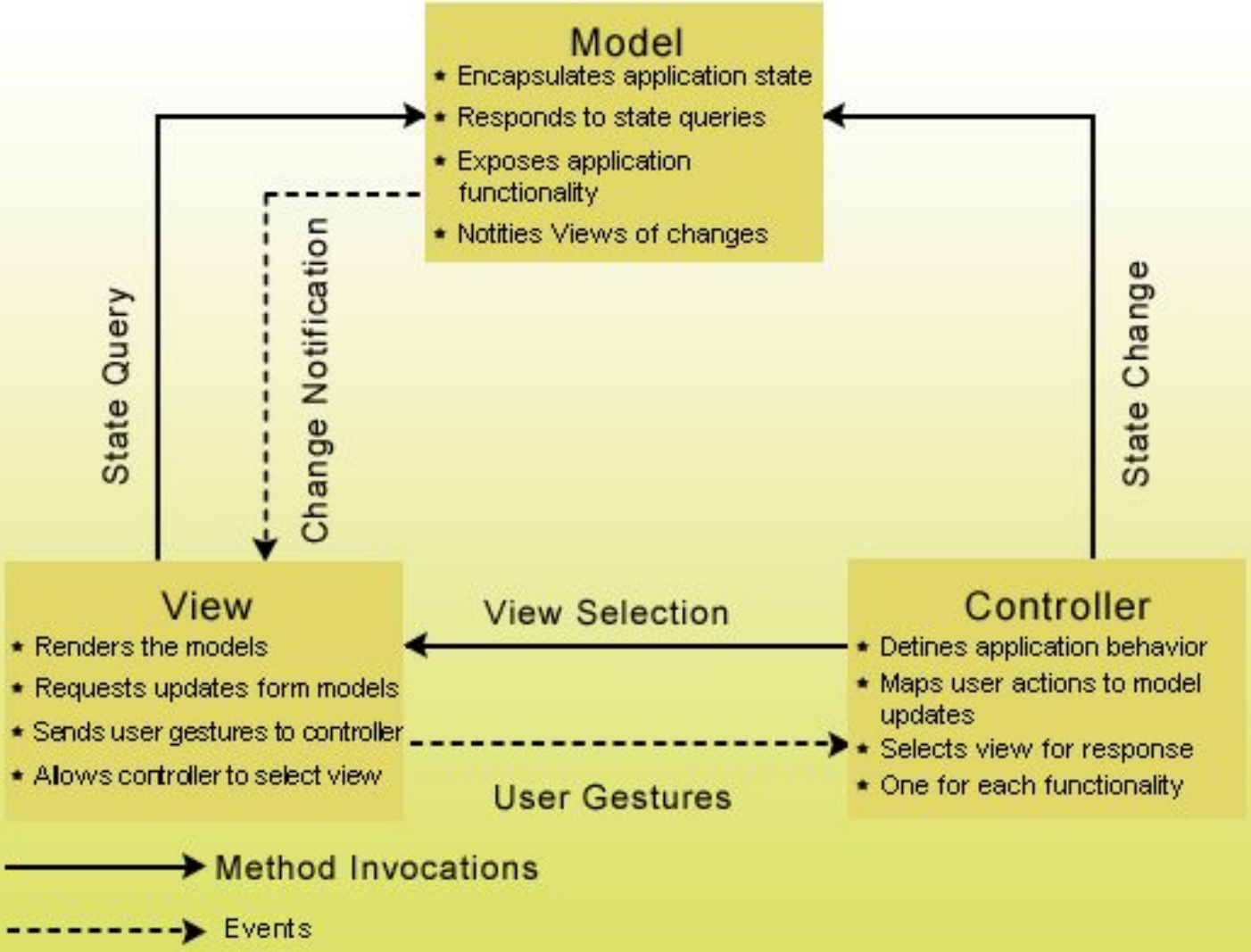
```
public bindTodoListChanged(callback: (todos: { id: number; text: string; complete: boolean }[]) => void): void {
  this.onTodoListChanged = callback;
}

public onTodoListChanged(todos: { id: number; text: string; complete: boolean }[]): void {
  // It will be overriden in run time by the Controller
}
```

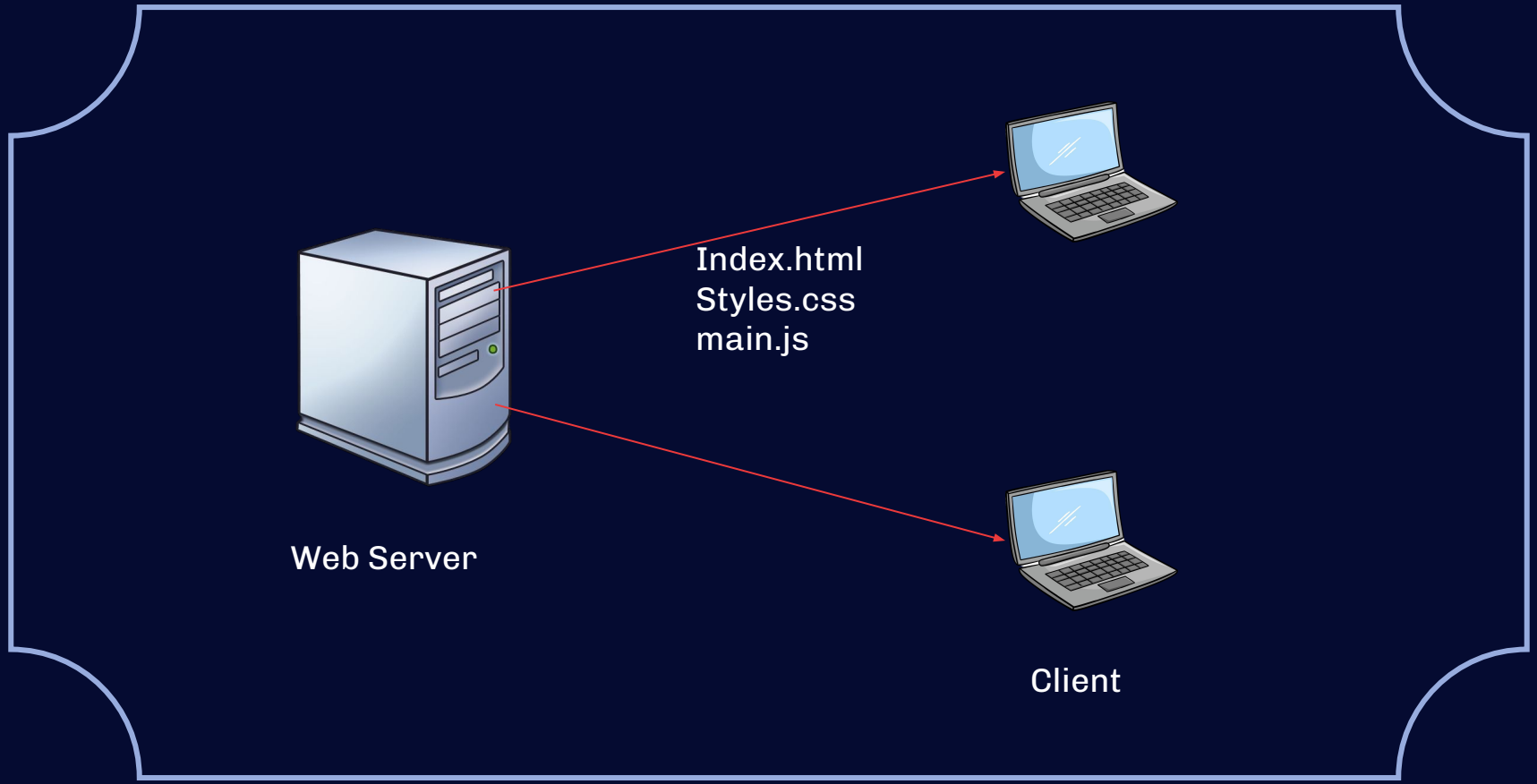
Controller

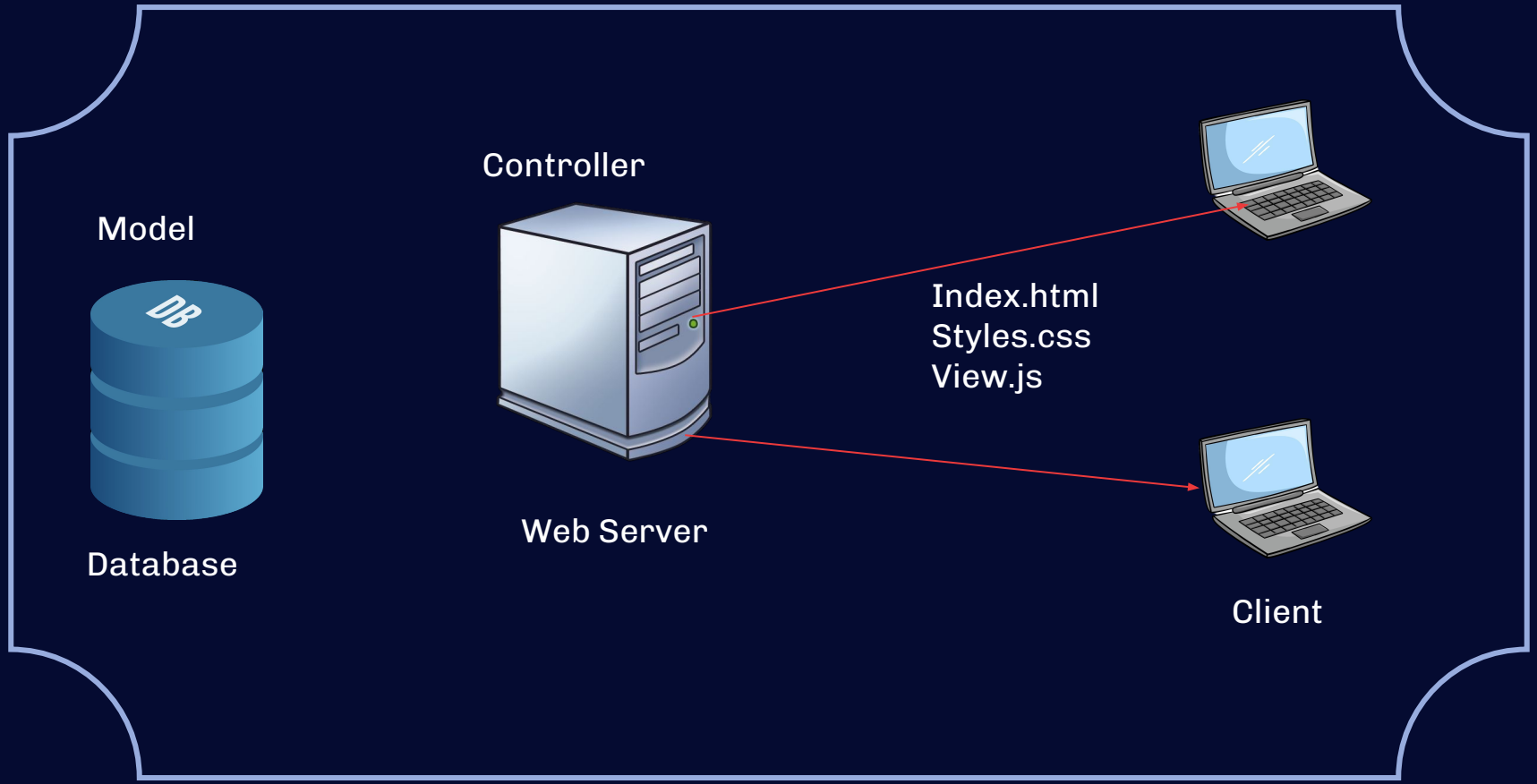
```
constructor(model: Model, view: View) {
  this.model = model;
  this.view = view;

  // Explicit this binding
  this.model.bindTodoListChanged(this.onTodoListChanged);
  this.view.bindAddTodo(this.handleAddTodo.bind(this));
  this.view.bindDeleteTodo(this.handleDeleteTodo.bind(this));
  this.view.bindToggleTodo(this.handleToggleTodo.bind(this));

  // Show initial todos
  this.onTodoListChanged(this.model.getTodos());
}
```

Model

```
public bindTodoListChanged(callback: (todos: { id: number; text: string; complete: boolean }[]) => void): void {
  this.onTodoListChanged = callback;
}

private onTodoListChanged(todos: { id: number; text: string; complete: boolean }[]): void {
  // This is a placeholder for a function that would be overwritten by the Controller
}

public addTodo(todoText: string): void {
  const todo = {
    id: this.todos.length > 0 ? this.todos[this.todos.length - 1].id + 1 : 1,
    text: todoText,
    complete: false,
  };

  this.todos.push(todo);

  this.onTodoListChanged(this.todos);
}
```


Model
Encapsulates application state
Responds to state queries
Exposes application functionality
Notities Views of changes
State Query
Change Notification
State Change
View
Renders the models
Requests updates form models
Sends user gestures to controller
Allows controller to select view
View Selection
User Gestures
Controller
Detines application behavior
Maps user actions to model updates
Selects view for response
One for each functionality
Method Invocations
Events


Index.html
Styles.css
main.js
Web Server
Client


Controller
Model
DB
Database
Web Server
Index.html
Styles.css
View.js
Client

Benefits

- Modularity
- Code reusability
- Software maintainability
- Scalability

Benefits

- Modularity

  → Collaborative work

  → Divide responsibilities

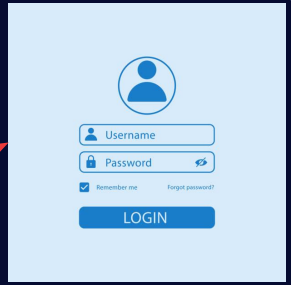
Benefits

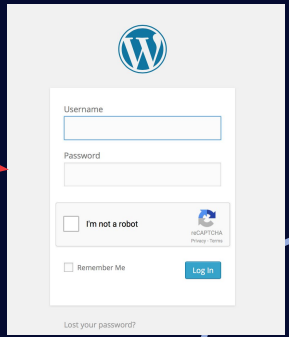
- Code reusability

→ Parts of the code can be reused

Example: Login functionality →

Controller





Benefits

- Software maintainability

→ Debugging

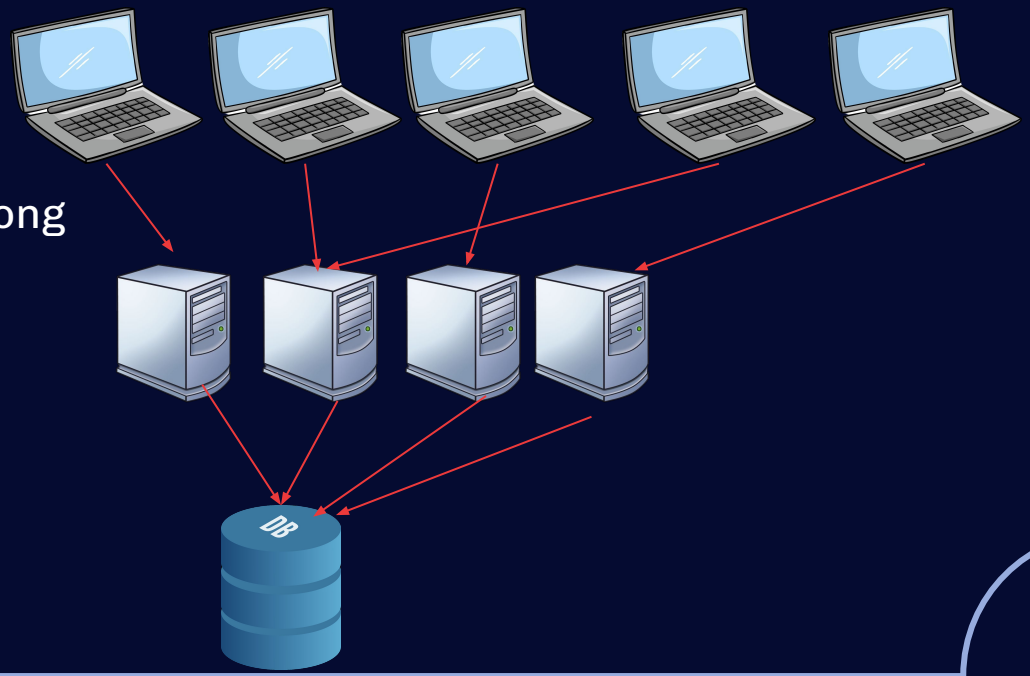
→ Adding new functionalities

→ Code understanding

Benefits

- Scalability

→ Distribute the load among different servers

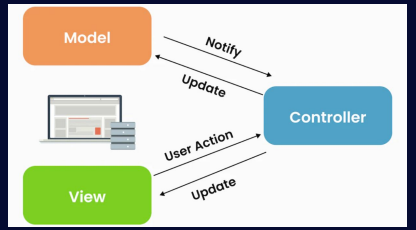


Frameworks for MVC

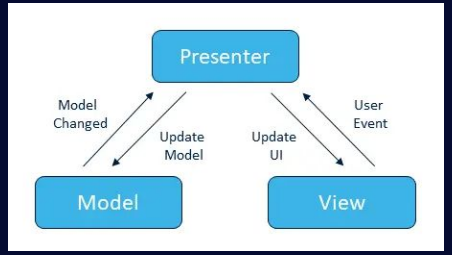
- ASP.NET MVC → C#
- Ruby on rails → Ruby
- Django → Python

Evolution of MVC

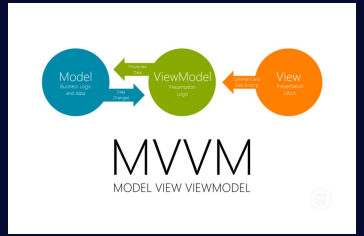
- MVC:



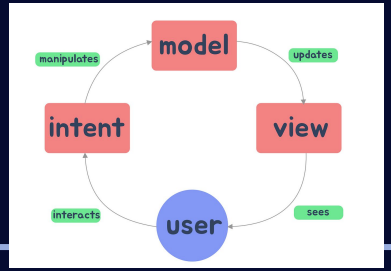
- MVP:



- MVVM

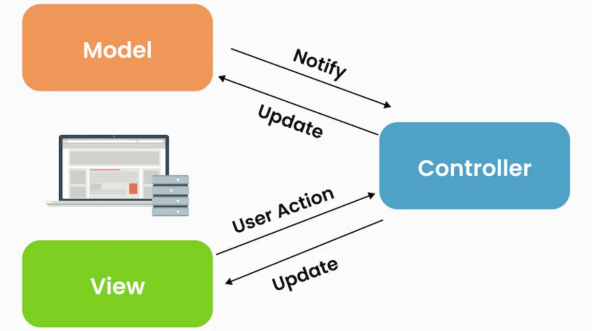


- MVI



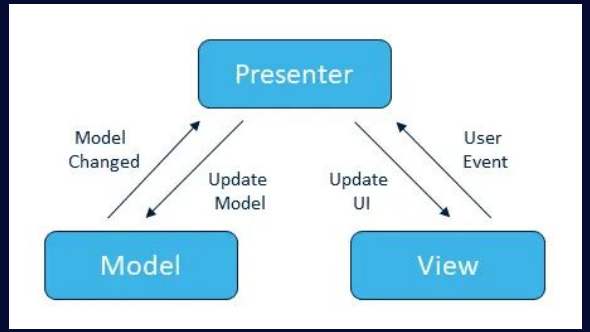
Evolution of MVC

- MVC:
  → clear separation between data (Model), user interface (View) and logic (Controller)
- Disadvantages
  → Issues with unit testing



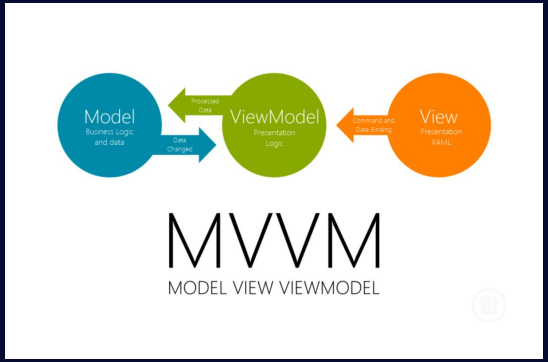
Evolution of MVC

- MVP:
  → Enables better testability and promotes the Single Responsibility Principle.
- Disadvantages
  → It can become complex as the project scales



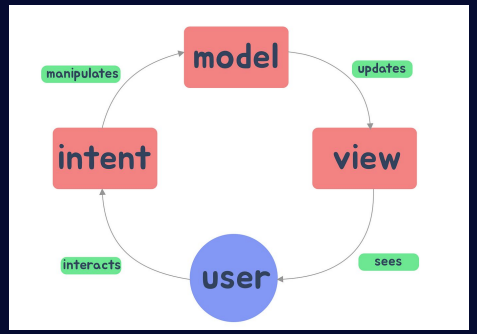
Evolution of MVC

- MVVM:
  → bidirectional data binding, making UI updates automatic and reducing boilerplate code.
- Disadvantages
  → Requires a good understanding of reactive programming concepts and data binding.



Evolution of MVC

- MVI:
  → Represents the actions or events that the user performs in the View.
- Disadvantages
  → It can introduce additional complexity, especially for simple projects.



Bibliography

- Book about programming with frameworks MVC

  https://www.c-sharpcorner.com/uploadfile/ebooks/11112013031641am/pdffile/programming%20asp.net%20mvc%205.pdf

- Java Model View Controller (MVC) Design Pattern

  https://www.roseindia.net/tutorial/java/jdbc/javamvcdesignpattern.html

- Implementation of an application following the MVC pattern in JavaScript

  https://www.taniarascia.com/javascript-mvc-todo-app/

Bibliography

- Origins of MVC

https://medium.com/@duncandevs/origins-of-model-view-controller-d685528857ce

- Evolution of MVC

https://medium.com/@KodeFlap/choosing-android-architectures-mvc-mvp-mvvm-clean-architecture-and-mvi-8ad2a43f7f9b