Chapter 7

Development with Framework

Outline

- Introduction to the MVC framework
- The model Layer
- The Controller Layer
- The View Layer
- Insert Record to a Database
- Update a Record in a Database
- Fetching Records from a Databases
- Deleting Record from a Database

Model-View-Controller (MVC) Architecture

- This presentation provides a comprehensive overview of the Model-View-Controller (MVC) architecture, a fundamental design pattern used in software engineering to separate application logic, data, and user interface.
- We'll also explore how CRUD operations integrate with MVC.
- Using MVC with CRUD operations ensures maintainable, scalable, and well-organized codebases.

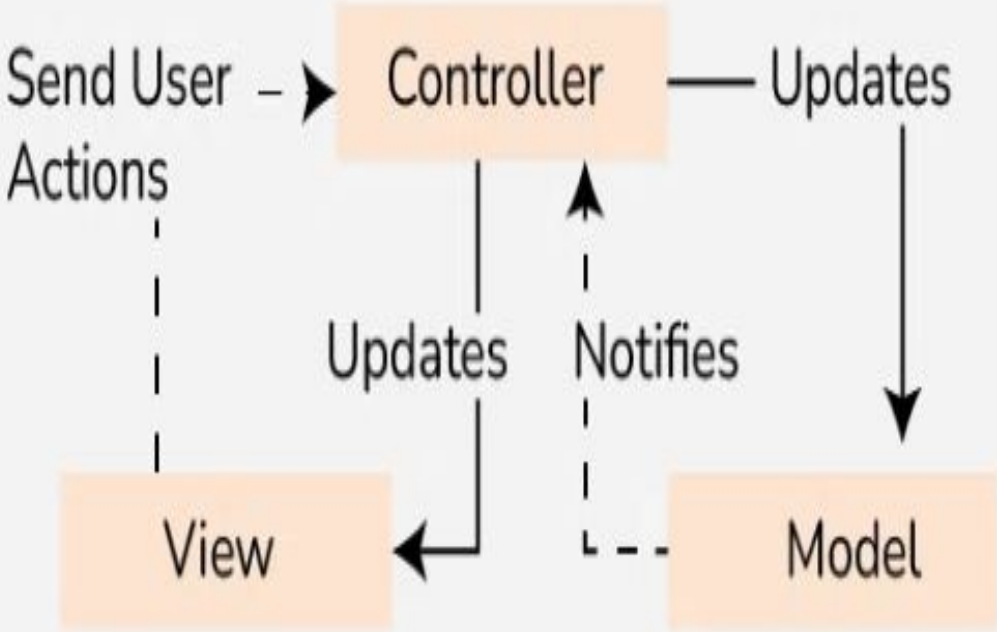
Introduction to the MVC Framework

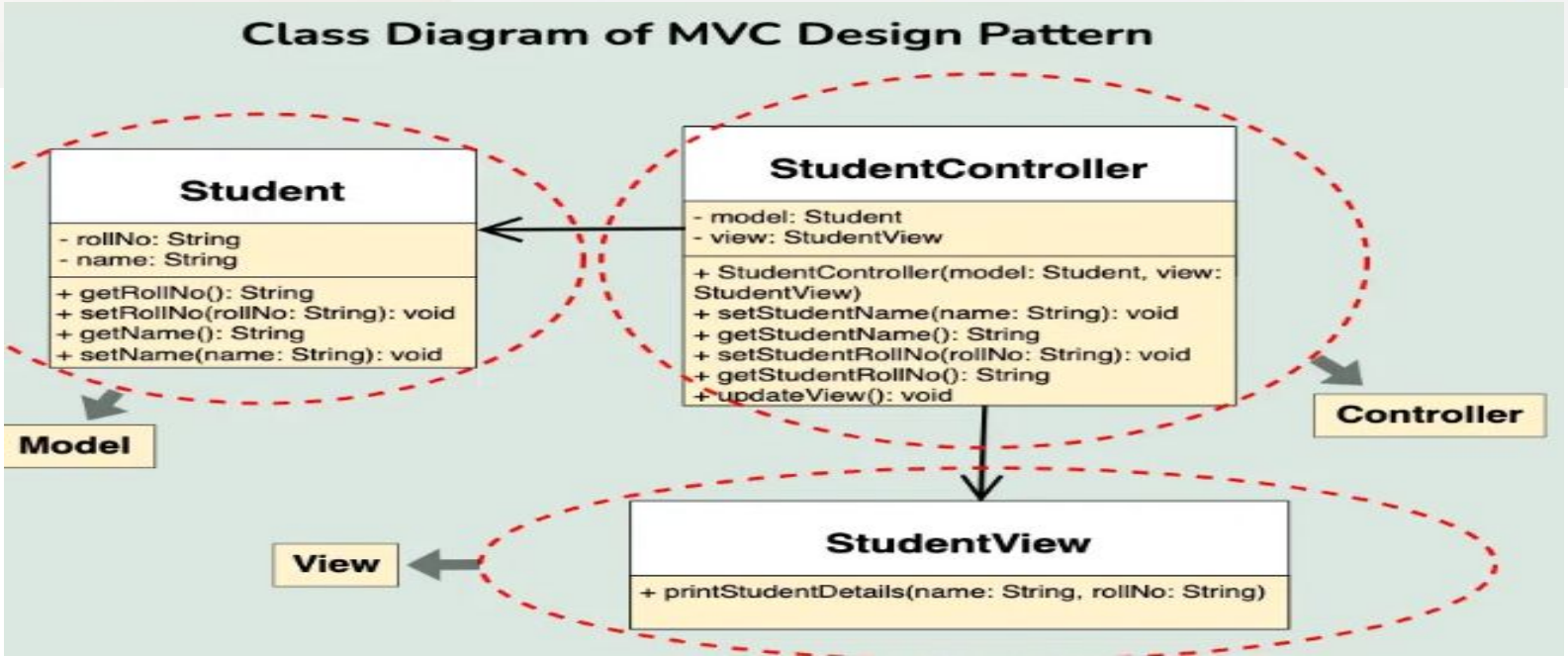
- The MVC framework divides an application into three interconnected components:

1. Model: Manages data and business rules.
2. View: Represents the UI layer, presenting data to the user.
3. Controller: Handles user input and updates the model and view accordingly.

This architecture promotes separation of concerns, making code easier to manage and scale.

MVC Design Pattern





Communication between the Components

- **User Interaction with View:** The user interacts with the View, such as clicking a button or entering text into a form.
- **View Receives User Input:** The View receives the user input and forwards it to the Controller.
- **Controller Processes User Input:** The Controller receives the user input from the View. It interprets the input, performs any necessary operations (such as updating the Model), and decides how to respond.
- **Controller Updates Model:** The Controller updates the Model based on the user input or application logic.
- **Model Notifies View of Changes:** If the Model changes, it notifies the View.
- **View Requests Data from Model:** The View requests data from the Model to update its display.
- **Controller Updates View:** The Controller updates the View based on the changes in the Model or in response to user input.
- **View Renders Updated UI:** The View renders the updated UI based on the changes made by the Controller.

The Model Layer

The Model is responsible for:

- Managing the application's data and logic.
- Interfacing with the database (e.g., fetching, inserting, updating records).
- Applying business rules and validations.

Example: A `User` model might have methods like `createUser()`, `getUserById()`, and `validateEmail()`.

Model (Student class)

Represents the data (student's name and roll number) and provides methods to access and modify this data

```
class Student {
   private $rollNo;
   private $name;

   public function getRollNo() {
      return $this->rollNo;
   }
   public function setRollNo($rollNo) {
      $this->rollNo = $rollNo;
   }
   public function getName() {
      return $this->name;
   }

   public function setName($name) {
      $this->name = $name;
   }
}
```

The Controller Layer

- The Controller acts as the intermediary between the Model and the View:

- Receives and processes user input.
- Calls Model functions to update or retrieve data.
- Selects the appropriate View to display results.

Example: A `UserController` might handle user registration by validating input, creating a user record, and redirecting to a welcome page.

View (StudentView class)

Represents how the data (student details) should be displayed to the user. Contains a method (**printStudentDetails**) to print the student's name and roll number.

```
<?php

class StudentView {
    public function printStudentDetails($studentName, $studentRollNo) {
        echo "Student:\n";
        echo "Name: " . $studentName . "\n";
        echo "Roll No: " . $studentRollNo . "\n";
    }
}

?>
```

The View Layer

- The View is the presentation layer responsible for displaying data:

- Renders HTML, CSS, and JavaScript to create the user interface.
- Gets data from the Controller.

Example: A user profile page shows data retrieved by the controller, rendered through an HTML template.

Controller (StudentController class)

Acts as an intermediary between the Model and the View. Contains references to the Model and View objects. Provides methods to update the Model (e.g., **setStudentName**, **setStudentRollNo**) and to update the View (**updateView**).

```
class StudentController {
    private $model;
    private $view;

    public function __construct(Student $model, StudentView $view) {
        $this->model = $model;
        $this->view = $view;
    }

    public function setStudentName($name) {
        $this->model->setName($name);
    }

    public function getStudentName() {
        return $this->model->getName();
    }

    // You can add other methods here as needed
}
```

Insert Record to a Database

In MVC:

- The user fills out a form (View).
- The Controller validates the data and calls the Model.
- The Model inserts the record into the database.

PHP Example:

```
$stmt = $conn->prepare("INSERT INTO users (name, email) VALUES (?, ?)");
$stmt->bind_param("ss", $name, $email);
$stmt->execute();
```

Update a Record in a Database

Steps for updating a record:

- The user edits a form (View).
- The Controller processes and validates the input.
- The Model updates the record in the database.

PHP Example:

```
$stmt = $conn->prepare("UPDATE users SET email=? WHERE id=?");
$stmt->bind_param("si", $email, $id);
$stmt->execute();
```

Fetching Records from a **Database**

Data retrieval follows this path:

- Controller requests data from the Model.
- Model fetches data from the database.
- Controller sends the data to the View for display.

PHP Example:

```
$result = $conn->query("SELECT * FROM users");
while($row = $result->fetch_assoc()) {
 echo $row["name"];
}
```

Deleting a Record from a Database

To delete a record:

- The user clicks a delete button (View).
- The Controller receives the request.
- The Model deletes the record from the database.

PHP Example:

```
$stmt = $conn->prepare("DELETE FROM users WHERE id=?");
$stmt->bind_param("i", $id);
$stmt->execute();
```

When to Use the MVC Design Pattern

- **Complex Applications**: Use MVC for apps with many features and user interactions, like e-commerce sites. It helps organize code and manage complexity.

- **Frequent UI Changes**: If the UI needs regular updates, MVC allows changes to the View without affecting the underlying logic.

- **Reusability of Components**: If you want to reuse parts of your app in other projects, MVC's modular structure makes this easier.

- **Testing Requirements**: MVC supports thorough testing, allowing you to test each component separately for better quality control.

When Not to Use the MVC Design Pattern

- **Simple Applications**: For small apps with limited functionality, MVC can add unnecessary complexity. A simpler approach may be better.

- **Real-Time Applications**: MVC may not work well for apps that require immediate updates, like online games or chat apps.

- **Tightly Coupled UI and Logic**: If the UI and business logic are closely linked, MVC might complicate things further.

- **Limited Resources**: For small teams or those unfamiliar with MVC, simpler designs can lead to faster development and fewer issues.

Model

talks to data source to retrieve and store data

Which database table is the requested data stored in?

What SQL query will get me the data I need?

View

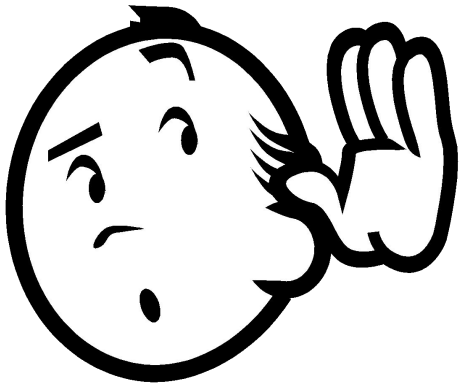
asks model for data and presents it in a user-friendly format

Would this text look better blue or red? In the bottom corner
or front and center?

Should these items go in a dropdown list or radio buttons?

Controller

listens for the user to change data or state in the UI, notifying the model or view accordingly



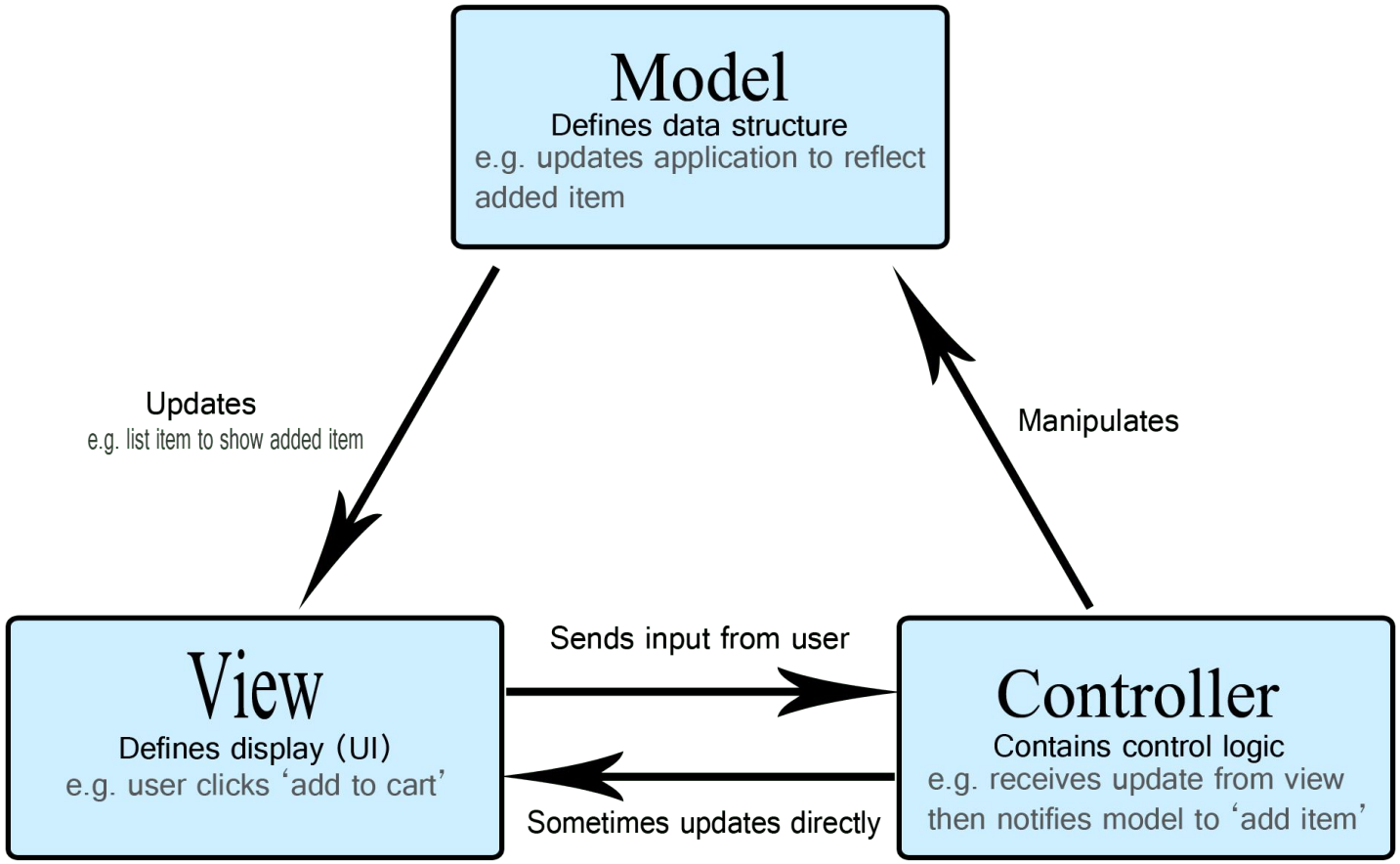
The user just clicked the "hide details" button. I better tell the view.

The user just changed the event details. I better let the model know to update the data.

Benefits of MVC

- Organization of code
  - Maintainable, easy to find what you need
- Ease of development
  - Build and test components independently
- Flexibility
  - Swap out views for different presentations of the same data (ex: calendar daily, weekly, or monthly view)
  - Swap out models to change data storage without affecting user

MVC Architecture



Push vs. Pull Architecture

- Push architecture
  - As soon as the model changes, it notifies all of the views
- Pull architecture
  - When a view needs to be updated, it asks the model for new data

Push vs. Pull Architecture

- Advantages for push
  - Guaranteed to have latest data in case something goes wrong later on
- Advantages for pull
  - Avoid unnecessary updates, not nearly as intensive on the view

MVC Example – Traffic Signal

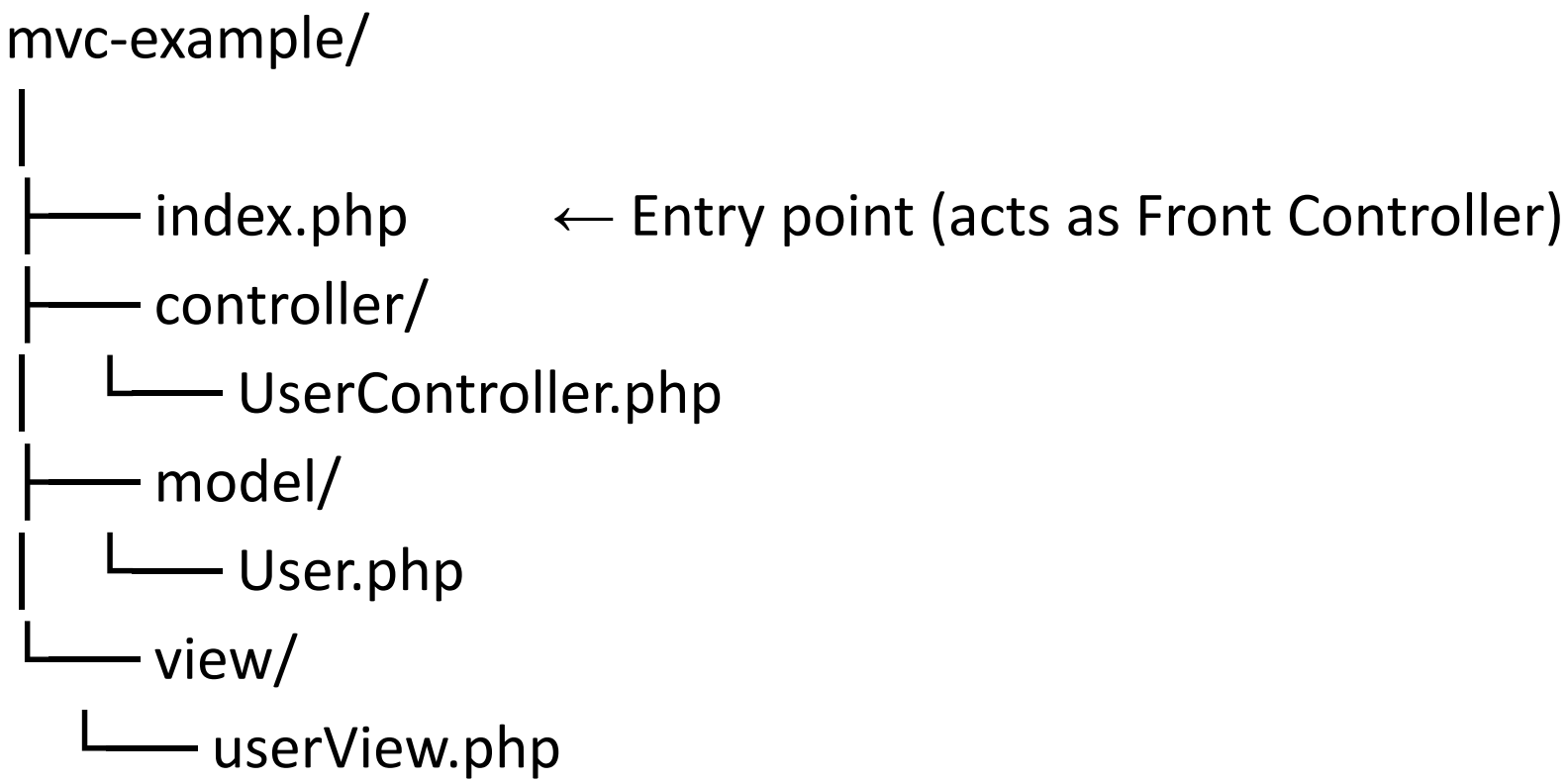
Traffic Signal – MVC

Component	Model	View	Controller
Detect cars waiting to enter intersection			X
Traffic lights to direct car traffic			
Regulate valid traffic movements		X	
Manual override for particular lights	X		
Detect pedestrians waiting to cross			X
Pedestrian signals to direct pedestrians			X
External timer which triggers changes at set interval		X	
			X

Traffic Signal

- **Model**
  - Stores current state of traffic flow
    - Knows current direction of traffic
    - Capable of skipping a light cycle
  - Stores whether there are cars and/or pedestrians waiting
- **View**
  - Conveys information to cars and pedestrians in a specific direction
- **Controller**
  - Aware of model's current direction
  - Triggers methods to notify model that state should change

More example



More example...

- index.php – Front Controller
- <?php
- require_once 'model/User.php';
- require_once 'controller/UserController.php';

- $controller = new UserController();
- $controller->showUser();

More example...

model/User.php – Model

```
<?php
class User {
    public function getUser() {
        return [
            'name' => 'John Doe',
            'email' => 'john@example.com'
        ];
    }
}
```

More example...

controller/UserController.php – Controller

```
<?php
require_once 'model/User.php';

class UserController {
    public function showUser() {
        $userModel = new User();
        $user = $userModel->getUser();

        // Pass data to view
        require 'view/userView.php';
    }
}
```

More example...

4. view/userView.php – View

```
<!DOCTYPE html>
<html>
<head>
   <title>User Info</title>
</head>
<body>
   <h1>User Information</h1>
   <p><strong>Name:</strong> <?php echo htmlspecialchars($user['name']);
?></p>
   <p><strong>Email:</strong> <?php echo htmlspecialchars($user['email']);
?></p>
</body>
</html>
```

Questions ??