**BRIEF EXPLANATION OF THE PURPOSE OF THE DOCUMENTATION**

The purpose of this documentation is to provide a comprehensive guide to building web applications using the PHP Core MVC architecture with PDO. This documentation aims to assist developers who are new to the PHP Core MVC architecture and want to build web applications that are fast, secure, and scalable. The documentation provides step-by-step instructions on how to build a web application using the PHP Core MVC architecture, along with sample code and best practices to follow.

**OVERVIEW OF PHP CORE MVC WITH PDO**

PHP Core MVC with PDO is a design pattern used to create web applications using PHP as the programming language. It separates the application into three main components: the Model, the View, and the Controller. The Model is responsible for data manipulation and retrieval, the View handles the presentation layer, and the Controller manages the flow of data between the Model and View.

PDO (PHP Data Objects) is a PHP extension used to manage database connections and execute SQL statements. It provides a simple and consistent API to interact with databases and supports a wide range of database drivers such as MySQL, PostgreSQL, SQLite, Oracle, and more.

This documentation aims to provide an overview of PHP Core MVC with PDO, its benefits, and how it can be used to develop fast, secure, and robust web applications. It covers the basic concepts of MVC, how to set up a PHP Core MVC with PDO application, and how to perform common tasks such as handling user authentication, input validation, and database operations.

**OVERVIEW OF PHP CORE MVC WITH PDO**

* Download the latest version of PHP and install it on your system.
* Download and install a web server like Apache or Nginx.
* Download and install a database management system like MySQL or MariaDB.
* Get the PHP Core MVC with PDO framework .
* Extract the downloaded archive and copy the contents to your web server directory.
* Create a new database in your database management system.
* Update .env file with the database credentials in this file with your own database credentials.
* Set up the virtual host and configure the web server to point to the public directory of the framework.
* Open your web browser and navigate to the virtual host URL to verify that the framework is working correctly.

**CREATING A NEW PROJECT**

To start a new PHP Core MVC project with PDO on your local machine, follow these steps:

* Install a local server environment such as XAMPP, WAMP, or MAMP. These software packages include Apache, MySQL, and PHP, which are the components required to run a PHP application locally.
* Once you have installed the local server environment, navigate to the htdocs folder (or the equivalent folder if using a different server environment) and create a new folder for your project.
* Copy the contents of the downloaded repository's "app" folder into your project folder. You may also copy the "public" folder, which contains the front controller file index.php and assets like CSS, JS and images.
* Create a new database for your project in phpMyAdmin, which is typically accessible at <http://localhost/phpmyadmin>. Make sure to note down the database name, username, and password, as you will need these later.
* Rename the ".env" file in your project folder to ".env" and update the database configuration variables to match your local database settings.
* Open a web browser and navigate to <http://localhost/your-project-directory> to see if the framework is installed and working correctly. If everything is set up correctly, you should see the welcome page of the PHP Core MVC framework.

**UNDERSTANDING THE LIBRARY MANAGEMENT SYSTEM PROJECT STRUCTURE**

The Library Management System includes features such as login, registration, forget password, admin dashboard, user dashboard, book inventory, borrowed book list, issue request, overdue book, user list with borrowed books, book request sent, view receipt and many more.

**CorePHP-PDO-MVC/**

**├── app/**

**│ ├── controllers/**

**│ │ ├── HomeController.php**

**│ │ └── ... (other controller files)**

**│ ├── models/**

**│ │ ├── UserModel.php**

**│ │ └── ... (other model files)**

**│ ├── views/**

**│ │ ├── home/**

**│ │ │ ├── index.php**

**│ │ │ └── ... (other view files for the home controller)**

**│ │ ├── layout/**

**│ │ │ ├── main.php**

**│ │ └── ... (other view directories)**

**│ ├── core/**

**│ │ ├── app.php**

**│ │ ├── controller.php**

**│ │ ├── database.php**

**│ │ ├── function.php**

**│ │ ├── language.php**

**│ │ ├── model.php**

**│ │ ├── pagination.php**

**│ │ ├── request.php**

**│ │ ├── response.php**

**│ │ ├── router.php**

**│ │ └── validator.php**

**│ └── init.php**

**│ └── .env**

**├── public/**

**│ ├── assets/**

**│ │ ├── css/**

**│ │ ├── js/**

**│ └── index.php**

**└── .htaccess**

**app/**: This directory contains the MVC components of the application such as controllers, models and views.

* **controllers/**: This directory contains all the controller files of the application.
* **models/**: This directory contains all the model files of the application.
* **views/**: This directory contains all the view files of the application organized by controller and action name.
* **core/**: This directory contains the core components of the application.
* **router.php**: This file contains the routing configuration of the application.
* **init.php**: This file is the main entry point of the application.
* **.env**: This file contains the environment variables for the application.

**public/**: This directory contains the publicly accessible files of the application such as css, js and images.

* **assets/**: This directory contains all the asset files of the application.
* **index.php**: This file is the front controller of the application.

**.htaccess**: This file contains the configuration for the Apache web server.

**EXPLANATION OF THE MODEL IN PHP CORE MVC**

The Model in PHP Core MVC is responsible for managing data and providing the necessary logic for the application. It is an essential component of the MVC architecture and separates the application's data and business logic from the user interface.

The Model contains methods to create, read, update and delete (CRUD) data from the database. It handles data validation, filtering, and sanitization. The Model is also responsible for establishing a connection to the database and executing queries.

In PHP Core MVC with PDO, the Model is often implemented using the PDO class, which provides an object-oriented interface for working with databases. This allows developers to write cleaner, more maintainable code by separating the database logic from the rest of the application.

**CONNECTING TO THE DATABASE USING PDO**

The topic of "Connecting to the Database Using PDO" is an essential aspect of PHP Core MVC. When developing an application, we need to store data in a database and retrieve it when required. PDO stands for PHP Data Objects, which is a PHP extension that provides an interface for accessing databases. It is a database abstraction layer that provides a consistent API to work with different types of databases.

The code snippet you have provided is a class named "Database" that uses PDO to connect to the database. The class has a constructor that sets up a new PDO instance with the database credentials, including the DB\_TYPE, DB\_HOST, DB\_NAME, DB\_USER, and DB\_PASS. The setAttribute() method is used to set the error mode to PDO::ERRMODE\_EXCEPTION, which throws an exception when an error occurs.

The class also includes two additional methods: getInstance() and getConnection(). The getInstance() method creates a new instance of the Database class if it doesn't already exist, ensuring that only one connection is made to the database. The getConnection() method returns the connection object created by the constructor.

To call this class, you would need to instantiate it in your application code by creating a new object of the Database class. Once you have an instance of the class, you can use the getConnection() method to get the connection object and execute database queries.

You can call database in model like this:

<?php

class AuthModel extends Model

{

    private $db;

    public function \_\_construct()

    {

      $this->db = Database::getInstance()->getConnection();

    }

    public function getUser($id){

        $stmt = $this->db->prepare("SELECT \* FROM td\_user WHERE id=:id");

        $stmt->execute([':id' => $id]);

        $user = $stmt->fetch();

        return $user;

    }

}

**PERFORMING CRUD OPERATIONS ON THE DATABASE**

Create (insert new data into database):

* booksbuild() - display the form to create a new book
* booksRecord(Request $request) - validate and save the new book data into the database

Read (fetch data from database):

* booksIndex()

- fetch all books data from database and display them in a paginated view

* booksDisplay(Request $request, $booksIdentify)

- fetch a single book data based on its unique identifier and display it

Update (modify existing data in database):

* booksModify(Request $request, $booksIdentify)

- display the form to edit an existing book data

* booksEdit(Request $request, $booksIdentify)

- validate and update the modified book data in the database

Delete (remove data from database):

* booksDestroy(Request $request, $booksIdentify)

- delete a book data from the database based on its unique identifier

**Explanation of the View in PHP Core MVC**

The views in this application are responsible for displaying data to the user. All view files should be placed in the **app/views** directory, organized by controller and action name.

**CREATING VIEWS USING HTML, CSS AND JAVASCRIPT**

All views should extend the Layout class located in **app/core/layout.php.** Views can be rendered by calling the render method of the Layout class and passing in the name of the **view file.**

<!DOCTYPE html>

<html>

<head>

    <meta charset="utf-8">

    <title><?php echo $title ?></title>

    <link rel="stylesheet" type="text/css" href="<?=ASSETS ?>css/style.css">

</head>

<body>

    <header>

    </header>

    <main>

        {{content}}

    </main>

    <footer>

        <p>&copy; <?php echo date("Y") ?> <?php echo APP\_NAME ?></p>

    </footer>

</body>

</html>

Every file under views will be implement **{{content}}** with this layout.php file

**RENDERING DATA FROM THE MODEL TO THE VIEW**

Here is a sample code to render data from model to view.

        $booksModel = $this->model('booksModel');

        $params['books'] =  $booksModel->displaySingle($booksIdentify);

        $this->view('books/booksSingle', $params);

Explanation of the Controller in PHP Core MVC

Controller plays a crucial role in handling user requests and responses. The Controller acts as the middleman between the Model and the View components of the application.

When a user makes a request, the Controller receives and processes it by invoking the appropriate Model methods to retrieve or update data from the database. The Controller then passes this data to the View component, which renders the response for the user.

In other words, the Controller manages the flow of data between the Model and the View. It retrieves data from the Model, processes it, and then sends it to the View to be displayed to the user.

The Controller also performs input validation and sanitization to ensure that the data received from the user is valid and secure. It communicates with the View to display error messages or success messages to the user.

PHP MVC/

        ├── app/

        │   ├── controllers/

        │   │   ├── HomeController.php

        │   │   └── ...

        │   ├── models/

        │   │   ├── User.php

        │   │   └── ...

        │   ├── views/

        │   │   ├── home/

        │   │   │   ├── index.php

        │   │   │   └── ...

        │   │   ├── layout/

        │   │   │   ├── main.php

        │   │   │   └── ...

        │   │   └── ...

        │   ├── core/

        │   │   ├── app.php

        │   │   ├── controller.php

        │   │   ├── database.php

        │   │   ├── function.php

        │   │   ├── language.php

        │   │   ├── model.php

        │   │   ├── pagination.php

        │   │   ├── request.php

        │   │   ├── response.php

        │   │   ├── router.php

        │   │   └── validator.php

        │   └── init.php

        │   └── .env

        ├── public/

        │   ├── assets/

        │   │   ├── css/

        │   │   ├── js/

        │   │   └── ...

        │   └── index.php (\*\*\* App Start having Router \*\*\*)

        └── .htaccess

To start php mvc app from public/index.php

<?php

    session\_start();

    include "../app/init.php";

    $app = new app(dirname(\_\_DIR\_\_));

  // declare router here

    $app->router->get('/welcome',function(){

        return "Welcome to Rapidrevert";

    });

    $app->run();

Here is a list of routers for example:

$app->router->get('/books', [BooksController::class, 'index']);

    $app->router->get('/books/build', [BooksController::class, 'build']);

    $app->router->post('/books/build', [BooksController::class, 'record']);

    $app->router->get('/books/{id}/destroy', [BooksController::class, 'destroy']);

    $app->router->get('/books/{id}/modify', [BooksController::class, 'modify']);

    $app->router->post('/books/{id}/modify', [BooksController::class, 'edit']);

    $app->router->get('/books/{id}', [BooksController::class, 'display']);

Or alternatively you can call only resources which will create above routers in single line of code for CRUD operation.

$app->router->resource('/path', 'model-prefix',ControllerName::class);

Connection to the database :

APP\_NAME=lms

APP\_PATH=http://localhostI211\_spring23/flabs/Library\_app/

    THEME=alpha-theme

    DB\_CONNECTION=mysql

    DB\_HOST=localhost

    DB\_DATABASE=library\_management

    DB\_USERNAME=phpuser

    DB\_PASSWORD=phpuser

Under app/init.php file we have files from app/core to run this app, If you want to add any feature or class is possible here:

<?php

    $\_ENV = parse\_ini\_file('.env', false, INI\_SCANNER\_RAW);

    define("APP\_NAME",$\_ENV["APP\_NAME"]);

    define('ROOT','http://localhost/your-app-directory);

    define('ASSETS', "<http://localhost/your-app-directory/public/assets/>");

    define('DB\_TYPE',$\_ENV["DB\_CONNECTION"]);

    define('DB\_HOST',$\_ENV["DB\_HOST"]);

    define('DB\_NAME',$\_ENV["DB\_DATABASE"]);

    define('DB\_USER',$\_ENV["DB\_USERNAME"]);

    define('DB\_PASS',$\_ENV["DB\_PASSWORD"]);

    include "../app/core/function.php";

    include "../app/core/app.php";

    include "../app/core/controller.php";

    include "../app/core/request.php";

    include "../app/core/response.php";

    include "../app/core/router.php";

    include "../app/core/database.php";

    include "../app/core/model.php";

    include "../app/core/validator.php";

    include "../app/core/pagination.php";

    include "../app/core/language.php";

**You can call database model like this:**

<?php

class QuoteModel{

    private $db;

    public function \_\_construct()

    {

     $this->db = Database::getInstance()->getConnection();

    }

    public function index(){

    }

}

To call a model in views or Controller

$BlogModel = $this->model('BlogModel');

To call a name parameter as url segment /{id} You can pass this like below:

    public function index(Request $request,$id){

        $BlogModel = $this->model('BlogModel');

        $data =  $BlogModel->displaySingle($id);

    }

Also, you can get input data from

public function index(Request $request){

$data = $request->getBody();

}

You can call validator class in your controller method like this:

$validator = new Validator();

        $validator->rules([

            'userName' => 'required|min:10',

            'userEmail' => 'required|max:250|email',

        ]);

        $validator->validate($data);

        if ($validator->fails()) {

            $errors = $validator->errors();

            foreach($errors as $error){

                    echo $error."</br>";

            }

        } else {

          // Call here model method to do your operation and give success message

        }

Here is View views/layout/main.php file :

<!DOCTYPE html>

<html>

<head>

    <meta charset="utf-8">

    <title><?php echo $title ?></title>

    <link rel="stylesheet" type="text/css" href="<?=ASSETS ?>css/style.css">

</head>

<body>

    <header>

    </header>

    <main>

        {{content}}

    </main>

    <footer>

        <p>&copy; <?php echo date("Y") ?> <?php echo APP\_NAME ?></p>

    </footer>

</body>

</html>