УПРАВЛЕНИЕ ЗАПРОСАМИ К БАЗЕ ДАННЫХ ПОЛЬЗОВАТЕЛЕЙ СОЦИАЛЬНОЙ СЕТИ ЧЕРЕЗ ПАТТЕРН MVC

Акиншева И. В., Пантюхов В. А.
Кафедра автоматизации технологических процессов и производств, Могилевский государственный университет продовольствия
Могилев, Республика Беларусь
E-mail: starrina@mail.ru

Разработанное сетевое приложение (социальная сеть) было спроектировано на архитектурном паттерне MVC (Model, View, Controller), который позволил расширить проект от локального узла до многопользовательской сетевой системы. Определен сценарий (механизм выработки) запросов. Для управления запросами использован язык PHP.

Введение

Используемый паттерн MVC, представляет собой схему разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер. Модификация каждого компонента осуществляется независимо [1].

Входной точкой в разработанное программное обеспечение, реализующее социальную сеть, является контроллер, который реагирует на действие пользователя и отправляет соответствующий ответ. Причем между этими двумя действиями контроллер может обратиться как к модели, так и к сервисам, которые делегируют выполнение работы по обработке данных моделям. Пользовательский интерфейс с точки зрения MVC называется View (Представление). Входная точка в приложение возвращает View. То, что каждый запрос возвращает View с инкасулированными в него данными, является не обязательным, это зависит от типа приложения.

Если это SPA (Single Page Application) приложение, то сервер возвращает View один раз, а все остальные запросы просто отправляют нужные данные клиентской части, которые самостоятельно генерирует View внутри себя (чаще всего с помощью фреймворка). Если же это SSR-SPA приложение (Server Side Rendering + Single Page Application), то сервер возвращает View больше одного раза, однако скрипты, формирующие запросы приложению, загружаются не сразу. Разработанное приложение имеет тип SPA. В качестве шаблона для построения серверной части выступает фреймворк Laravel.

I. Основная часть

Используемая в приложении гипертекстовая разметка является минимальной с точки зрения формирования базовой html-структуры.

Запросы происходят по следующему сценарию:

- пользователь входит в приложение, основной контроллер принимает запрос от приложения «клиента», расположенного на машине пользователя;
- контроллер возвращает приложению «клиента» файл с гипертекстовой разметкой (view) и скрипты;
- браузер отображает файл с гипертекстовой разметкой и загружает скрипты;
- при загрузке скриптов в приложение «клиента» оформляется остальная часть пользовательского интерфейса и добавляются элементы интерактивности на страницу в браузере.

Выполнение такой последовательности действий, позволяет пользователю увидеть свой аккаунт в социальной сети как можно быстрее.

Из описанного выше следует, что используемый паттерн MVC является универсальным, не зависит от подхода к разработке приложения и позволяет оперативно изменять конфигурацию приложения.

В данном проекте паттерн MVC реализован полностью, все три сущности изолированы друг от друга благодаря использованию современного фреймворка Laravel, написанного на языке PHP.

Рассмотрим, как реализован паттерн MVC в данном приложении (рис. 1).

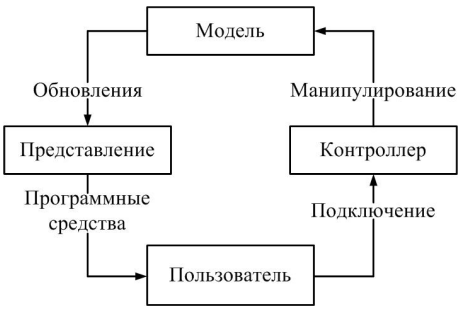

Рис. 1 – Диаграмма MVC, используемая для формирования приложения

В первую очередь запрос попадет в файл routes/web.php, который является точкой входа для любого запроса в данном приложении. Файл содержит имена всех роутеров и контроллеров,

которые отвечают за используемые и выполняемые запросы.

В разработанном приложении посылается запрос по корневому адресу приложения, за который отвечает представленный ниже элемент файла web.php:

Route::get('/', function () { //Главная страница

return view('welcome'); // показать файл welcome.blade.php

})->name('home'); // Имя роутера home

В данном случае не описывается, какой конкретно контроллер отвечает за логику обработки запроса, поскольку запрос обрабатывается прямо внутри представленного блока кода, возвращая «view» с именем «welcome». Интерпретатор воспринимает, что полное название файла будет welcome.blade.php [2].

Созданный файл welcome.blade.php является так называемым «view engine». Это обычный файл с расширением php, который интерпретатор превращает на выходе в файл с форматом .html, предварительно вставив в него динамическую информацию и сгенерировав динамическую html-структуру [3]. Например, для пользователя, написавшего два сообщения будет сгенерирована структура, содержащая два html-блока с различным содержанием, но одинаковой html-структурой. После формирования окончательной html-структуры, интерпретатор возвращает файл с расширением .html и сервер возвращает данный файл приложению «клиента».

Рассмотрим для примера один из контроллеров, реализованных в разработанном приложении. Сам файл представляет собой класс, наследующий класс фреймворка Laravel, который реализует различные интерфейсы. Класс UserController инкапсулирует в себе публичные методы, задающие логику обработки запросов, привязанных конкретно к используемому контроллеру. Этот механизм реализован в файле web.php. Пусть путь к post-запросу прописывается следующим образом: «socialnet.loc/signin». Этот post-запрос предназначен для авторизации пользователя. Запрос поступает в файл web.php, где обрабатывается следующим блоком кода:

Route::post('/signin', [

'uses' => 'UserController@postSignIn',

'as' => 'signIn']);

Во второй строке кода реализована логика, задающая контроллер конкретно для этого запроса. Также делегируется возможность использования метода ответственным за обработку запроса типа post. Контроллер с именем UserController выполняет метод postSignIn. «As» используется для сокращенного обращения к классу UserController через view. Используя сокращение, появляется возможность активировать этот контроллер через любое view следующим образом:

<form action="{{ route('signIn') }}"method ="post» <html код> </form>

Данная строка обрабатывается компилятором и на выходе получается следующий код:

<form action="/signin"method="post» <html код> </form>

В общем случае считается хорошей практикой использовать такого рода сокращения методов. Запросы могут быть длинными, а код желательно изначально делать однотипным в пределах всего приложения. Кроме того в любом месте кода можно активировать данный запрос в выражении «route('signIn')».

При создании модели данных, в представленном случае это будет пользователь, необходимо создать для нее класс и установить связи с другими моделями данных. В данном приложении используются связи «один ко многим», «многие ко многим», «один к одному». Рассмотрим пример, когда пользователь имеет несколько постов. На программном уровне это инициализируется следующим образом:

class User extends Model implements Authenticatable {

public function posts() {

return $this->hasMany('AppPost');

//Relations!! }}

Реализация связи «многие к одному» приводит к тому, что пользователь может иметь несколько постов, в свою очередь пост имеет одного пользователя:

class Post extends Model {

public function user() {

return $this->belongsTo('AppUser'); }}

Согласно приведенному коду, при компиляции автоматически создаются таблицы в базе данных с требуемыми связями. Таким образом, разработчику не нужно знать все нюансы синтаксиса SQL для создания связей между отдельными таблицами в базе данных.

II. Заключение

В результате было разработано сетевое приложение на архитектурном паттерне MVC со следующими возможностями: валидация данных на стороне «сервера»; валидация данных на стороне «клиента»; применение современных UI-стандартов со стороны «клиента»; защищенное соединение с административной панелью, расширенные возможности работы с интерфейсом страницы пользователя.

1. Стивенс, У. Р. UNIX: разработка сетевых приложений / У. Р. Стивенс, Б. Феннер, Э. М. Рудофф. –СПб.: Питер, 2007. –1039 с.
2. Котеров, Д. PHP 7 / Д. Котеров, И. Симдянов. – СПб.: БХВ-Петербург, 2016. –1088 с.
3. Бибо, Б. jQuery. Подробное руководство по JavaScript / Б. Бибо, И. Кац; пер. с англ. – СПб.: Символ-Плюс, 2011. –624 с.