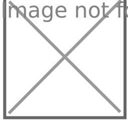
Содержание:



Введение

MVC — это шаблон программирования, который позволяет разделить логику приложения на три части:

- *Model* (модель). Получает данные от контроллера, выполняет необходимые операции и передаёт их в вид.
- *View* (вид или представление). Получает данные от модели и выводит их для пользователя.
- *Controller* (контроллер). Обрабатывает действия пользователя, проверяет полученные данные и передаёт их модели.

Основная цель применения этой концепции программирования состоит в отделении бизнес-логики (модели) от её визуализации (представления, вида). За счёт такого разделения повышается возможность повторного использования кода. Наиболее полезно применение данной концепции в тех случаях, когда пользователь должен видеть те же самые данные одновременно в различных контекстах и/или с различных точек зрения. В частности, выполняются следующие задачи:

1. К одной *модели* можно присоединить несколько *видов*, при этом не затрагивая реализацию *модели*. Например, некоторые данные могут быть одновременно представлены в виде электронной таблицы, гистограммы и круговой диаграммы;
2. Не затрагивая реализацию *видов*, можно изменить реакции на действия пользователя (нажатие мышью на кнопке, ввод данных) — для этого достаточно использовать другой *контроллер*;
3. Ряд разработчиков специализируется только в одной из областей: либо разрабатывают графический интерфейс, либо разрабатывают бизнес-логику. Поэтому возможно добиться того, что программисты, занимающиеся разработкой бизнес-логики (*модели*), вообще не будут осведомлены о том, какое *представление* будет использоваться.

Как работает MVC

Пользователь направляет запрос в контроллер (в случае веб-приложений – это обращение по адресу), контроллер (Controller) обрабатывает запрос, запрашивает данные от соответствующих моделей (Model), получает данные, может быть, выполняет какую-то дополнительную их обработку, например, агрегирует их с другими данными и затем передает данные в представление (View). Представление формирует данные в соответствии с заданным шаблоном отображения и возвращает результат пользователю. Штрих-пунктирной линией показано опосредованное взаимодействие представления с контроллером через AJAX-сценарии и POST-запросы.

Логика работы приложения построенного на основе MVC-архитектуры представлена на рисунке 1.

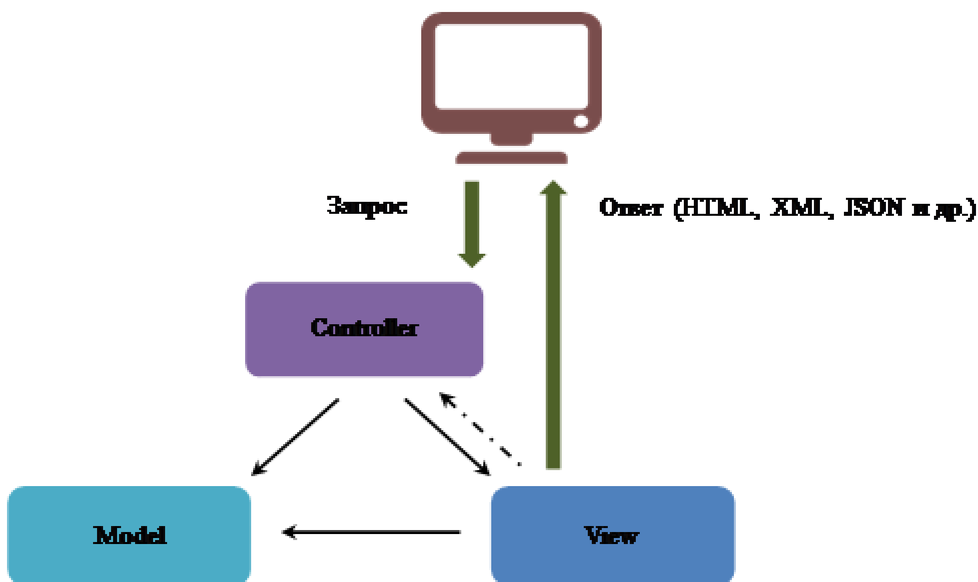


Рис.1

Функциональные возможности и расхождения

Поскольку MVC не имеет строгой реализации, то реализован он может быть по-разному. Нет общепринятого определения, где должна располагаться бизнес-логика. Она может находиться как в контроллере, так и в модели. В последнем случае, модель будет содержать все бизнес-объекты со всеми данными и

функциями.

Некоторые фреймворки жестко задают где должна располагаться бизнес-логика, другие не имеют таких правил.

Также не указано, где должна находиться проверка введённых пользователем данных. Простая валидация может встречаться даже в представлении, но чаще они встречаются в контроллере или модели.

Интернационализация и форматирование данных также не имеет четких указаний по расположению.

Концепция MVC позволяет разделить модель, представление и контроллер на три отдельных компонента:

Модель

Модель предоставляет данные и методы работы с ними: запросы в базу данных, проверка на корректность. Модель не зависит от представления (не знает как данные визуализировать) и контроллера (не имеет точек взаимодействия с пользователем) , просто предоставляя доступ к данным и управлению ими.

Модель строится таким образом, чтобы отвечать на запросы, изменяя своё состояние, при этом может быть встроено уведомление «наблюдателей».

Модель, за счёт независимости от визуального представления, может иметь несколько различных представлений для одной «модели».

Представление

Представление отвечает за получение необходимых данных из модели и отправляет их пользователю. Представление не обрабатывает введённые данные пользователя.

Контроллер

Контроллер обеспечивает «связь» между пользователем и системой. Контролирует и направляет данные от пользователя к системе и наоборот. Использует модель и представление для реализации необходимого действия.

Условно-обязательные модификации

Для реализации схемы «Model-View-Controller» используется достаточно большое число шаблонов проектирования (в зависимости от сложности архитектурного решения), основные из которых — «наблюдатель», «стратегия», «компоновщик».

Наиболее типичная реализация — в которой представление отделено от модели путём установления между ними протокола взаимодействия, использующего «аппарат событий» (обозначение «событиями» определённых ситуаций, возникающих в ходе выполнения программы, — и рассылка уведомлений о них всем тем, кто подписался на получение): при каждом особом изменении внутренних данных в модели (обозначенном как «событие»), она оповещает о нём те зависящие от неё представления, которые подписаны на получение такого оповещения — и представление обновляется. Так используется шаблон «наблюдатель».

При обработке реакции пользователя — представление выбирает, в зависимости от реакции, нужный контроллер, который обеспечит ту или иную связь с моделью. Для этого используется шаблон «стратегия», или вместо этого может быть модификация с использованием шаблона «команда».

Для возможности однотипного обращения с подобъектами сложно-составного иерархического вида — может использоваться шаблон «компоновщик». Кроме того, могут использоваться и другие шаблоны проектирования — например, «фабричный метод», который позволит задать по умолчанию тип контроллера для соответствующего вида.

Чтение баз данных

Чтение данных из базы стоит ограничить, рассмотрим две стратегии, которыми здесь можно руководствоваться.

1. **Семантическая стратегия** – свести к минимуму или полностью исключить все операции, явно отражающие в своей сигнатуре специфику обращения к БД.   С точки зрения данной стратегии такой запрос в представлении неприемлем:
$postponements = Postponement::find()

->andWhere(['type_id' => Postponement::TYPE_TASK])

->andWhere(['object_id' => $task_id])

```
->asArray()

->all();
```

Здесь явная зависимость от БД, от специфики обращения к ней (пусть даже и средствами ORM). Однако тот же самый запрос, оформленный как метод модели, вполне приемлем:

```
class Task extends \yii\db\ActiveRecord { public function getPostponements() {     return Postponement::find()

->andWhere(['type_id' => Postponement::TYPE_TASK])

->andWhere(['object_id' => $this->id])

->all();

}

…

}
```

Использование метода в представлении:

```
$postponements = $task->postponements;
```

Общая идея данной стратегии заключается в том, чтобы абстрагировать методы обращения к БД, скрыть их от представления. Представление оперирует моделью предметной области, ее концептуальным смыслом, а не техническими аспектами реализации. В вышеприведенном примере $task->postponements предоставляет переносы (записи о том, когда и кем были осуществлены переносы сроков выполнения задачи). Представление будет отображать информацию, но каким образом эта информация сформирована, представлению знать не нужно.

2. **Стратегия запросов** – свести к минимуму или полностью исключить все SQL-запросы к БД из представления. В рамках этой стратегии мы инициируем все обращения к БД в контроллере и передаем полученные данные в представление, т.е. представление оперирует данными из оперативной памяти. Если в представление передается коллекция объектов модели имеющей агрегированные данные, отражающие реляционную связь с другими моделями, то необходимо осуществить жадную загрузку (eager loading). Возможность такой загрузки

реализована в современных ORM-решениях (и в Yii и в Laravel она есть). Исключение всех операций из представления, инициирующих SQL-запросы – это более строгий по сравнению с первой стратегией подход. Здесь мы следуем принципу единичной ответственности в представлении буквально: только отображение имеющихся данных и ничего сверх этого. Методы модели могут быть вызваны только в том случае, если они не генерируют SQL-запросы. Помимо преимуществ чисто методологического плана (соблюдение принципа единичной ответственности) такой подход привлекателен с точки зрения управления кэшированием данных. В рамках этой стратегии все данные проходит через контроллер, что существенно облегчает проверку их наличия в кэше и сохранения, если данные отсутствуют. Альтернатива – перекладывание этой работы на модель, что, естественно, не самая удачная идея, т.к. ответственность модели неоправданно расширяется. Кэширование же на уровне контроллера вполне естественно и органично.

Ошибки новичков

Начинающие программисты очень часто трактуют архитектурную модель MVC как пассивную модель. MVC: модель выступает исключительно совокупностью функций для доступа к данным, а контроллер содержит бизнес-логику. В результате — код моделей по факту является средством получения данных из СУБД, а контроллер — типичным модулем, наполненным бизнес-логикой. В результате такого понимания — MVC-разработчики стали писать код, который Pádraic Brady (известный в кругах сообщества «Zend Framework») охарактеризовал как «ТТУК» («Толстые, тупые, уродливые контроллеры»; Fat Stupid Ugly Controllers):

Среднестатистический ТТУК получал данные из БД (используя уровень абстракции базы данных, делая вид, что это модель) или манипулировал, проверял, записывал, а также передавал данные в Представление. Такой подход стал очень популярен потому, что использование таких контроллеров похоже на классическую практику использования отдельного php-файла для каждой страницы приложения.

Но в объектно-ориентированном программировании используется активная модель MVC, где модель — это не только совокупность кода доступа к данным и СУБД, но и вся бизнес-логика; также, модели могут инкапсулировать в себе другие модели. Контроллеры же, — как элементы информационной системы, — ответственны лишь за:

- приём запроса от пользователя;
- анализ запроса;

выбор следующего действия системы, соответственно результатам анализа (например, передача запроса другим элементам системы).

Только в этом случае контроллер становится «тонким» и выполняет исключительно функцию связующего звена (glue layer) между отдельными компонентами информационной системы.

Преимущества

Прежде всего дифференциация разработчиков php сайта на отделы. Также увеличивается скорость работы php приложения, если создается крупный проект. Ну и то, что касается непосредственно самого php разработчика, это правильная структуризация php кода (все на своих местах, так легче для понимания).

Список используемой литературы

https://ru.wikipedia.org/wiki/Model-View-Controller#Концепция

https://moluch.ru/archive/87/16899/