Подход MVC

JSP & Servlets
14 уровень, 2 лекция

ОТКРЫТА

Знакомство с архитектурой MVC

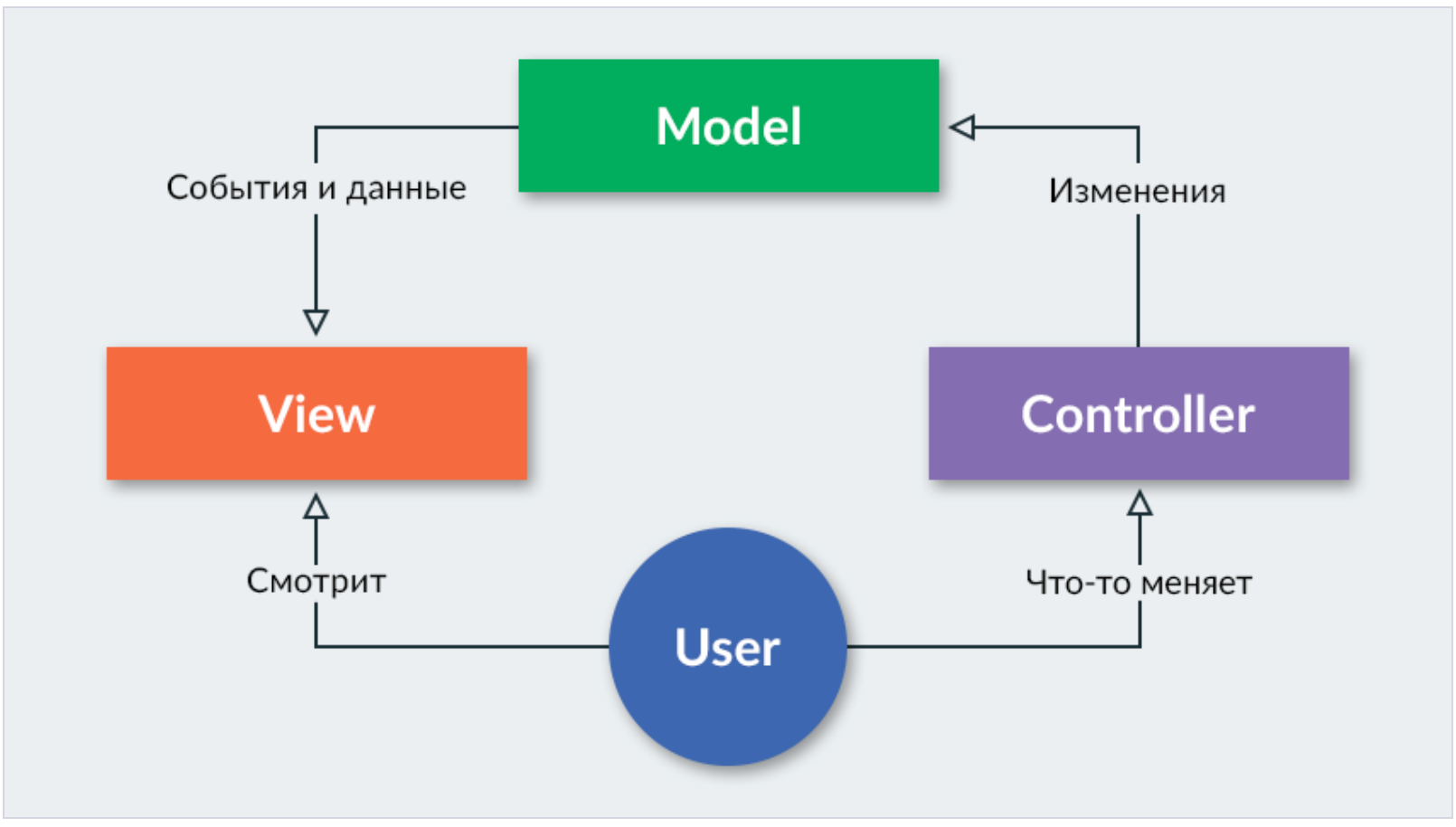
Самая популярная архитектура приложений, о которой знает каждый программист, — это **MVC**. MVC расшифровывается как **Model-View-Controller**.

Это не столько архитектура приложений, как архитектура компонентов приложения, но к этому нюансу вернемся попозже. Что же такое MVC?

MVC — это схема разделения данных приложения и управляющей логики на три отдельных компонента: **модель, представление и контроллер** — таким образом, что модификация каждого компонента может осуществляться независимо.

- **Модель (Model)** предоставляет данные и реагирует на команды контроллера, изменяя свое состояние.
- **Представление (View)** отвечает за отображение данных модели пользователю, реагируя на изменения модели.
- **Контроллер (Controller)** интерпретирует действия пользователя, оповещая модель о необходимости изменений.

Эту модель придумали еще в 1978 (!) году. Да, проблемы с правильной архитектурой ПО были актуальны еще 50 лет назад. Вот как эта модель описывается диаграммой в оригинале:



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

Модель строится таким образом, чтобы отвечать на запросы, изменяя свое состояние, при этом может быть встроено уведомление “наблюдателей”. Модель, за счет независимости от визуального представления, **может иметь несколько различных представлений** для одной “модели”.

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

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.

Определенная сложность есть с тем, что данная модель за десятки лет немного эволюционировала. То есть название осталось тем же, а назначение частей начало меняться.

Архитектура MVC в вебе

Идея, которая лежит в основе конструкционного шаблона MVC, очень проста: нужно четко разделять ответственность за различное функционирование в наших приложениях:

`Model` — обработка данных и логика приложения.

`View` — предоставления данных пользователю в любом поддерживаемом формате.

`Controller` — обработка запросов пользователя и вызов соответствующих ресурсов.

Приложение разделяется на три основных компонента, каждый из которых отвечает за различные задачи. Давай подробно разберем компоненты клиент-серверного приложения на примере.

Контроллер (Controller)

Пользователь кликает на различные элементы на странице в браузере, в результате чего браузер отправляет различные HTTP запросы: GET, POST или другие. К контроллеру можно отнести браузер и JS-код, которые работают внутри страницы.

Основная функция контроллера в данном случае — это вызывать методы у нужных объектов, управлять доступом к ресурсам для выполнения задач, заданных пользователем. Обычно контроллер вызывает соответствующую модель для задачи и выбирает подходящий вид.

Модель (Model)

Модель в широком смысле — это данные и правила, которые используются для работы с данными — вместе они составляют бизнес-логику приложения. Проектирование приложения всегда начинается с построения моделей объектов, которыми оно оперирует.

Допустим, у нас есть интернет-магазин, который торгует книгами, тогда человек — это только пользователь приложения или еще и автор книги? Эти важные вопросы должны быть решены во время проектирования модели.

Дальше идут наборы правил: что можно делать, что нельзя, какие наборы данных допустимы, а какие нет. Может ли книга быть без автора? А автор без книг? Может ли дата рождения пользователя быть в 300 году и тому подобное.

Модель дает контроллеру представление данных, которые запросил пользователь (сообщение, страницу книги, картинки, и тому подобное). Модель данных будет одинаковой, вне зависимости от того, как мы хотим представлять их пользователю. Поэтому мы выбираем любой доступный вид для отображения данных.

Модель содержит наиболее важную часть логики нашего приложения, логику, которая решает задачу, с которой мы имеем дело (форум, магазин, банк и тому подобное). Контроллер содержит в основном организационную логику для самого приложения (прямо как твой Project Manager).

Вид (View)

View обеспечивает различные способы представления данных, которые получены из модели. Он может быть шаблоном, который заполняется данными. Может быть несколько различных views и контроллер выбирает, какой подходит наилучшим образом для текущей ситуации.

Веб-приложение обычно состоит из набора контроллеров, моделей и видов (views). Контроллер может быть только на бэкенде, но также может быть вариант нескольких контроллеров, когда его логика размазывается и по frontend’у тоже. Хороший пример такого подхода — любое мобильное приложение.

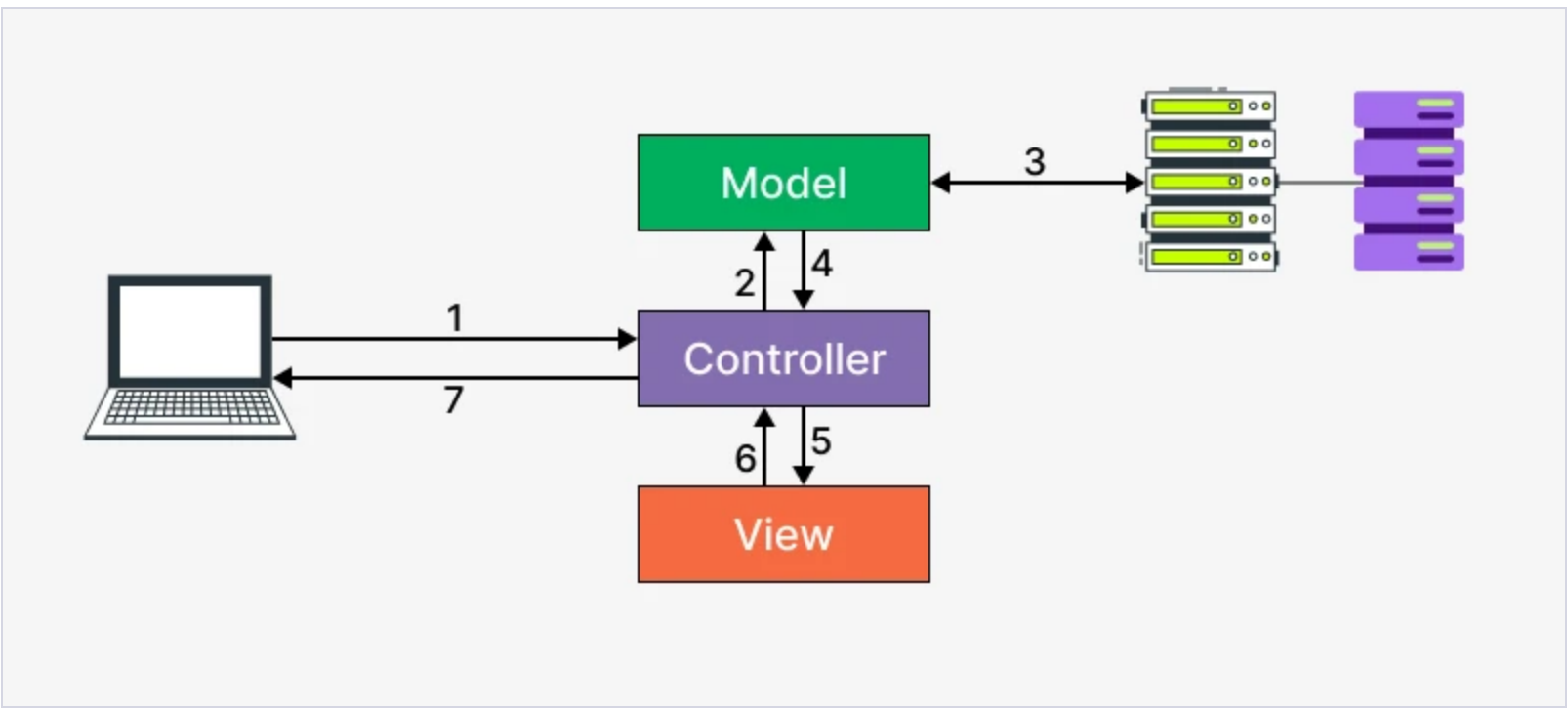
Пример MVC в вебе

Допустим тебе нужно разработать интернет-магазин, который будет заниматься продажей книг. Пользователь может выполнять следующие действия: просматривать книги, регистрироваться, покупать, добавлять пункты к текущему заказу, отмечать понравившиеся книги и покупать их.

В твоем приложении должна быть **модель**, которая отвечает за всю бизнес-логику. Также нужен **контроллер**, который будет обрабатывать все действия пользователей и превращать их в вызовы методов из бизнес-логики. При этом один метод контроллера может вызвать много различных методов модели.

Также тебе нужны наборы views: список книг, информация об одной книге, корзина, список заказов. Каждая страница веб-приложения — это фактически и есть отдельный view, который отображает пользователю определенный аспект модели.

Давай посмотрим, что произойдет, если пользователь откроет список рекомендованных магазином книг для просмотра названий. Всю последовательность действий можно описать в виде 6 шагов:



Шаги:

1. Пользователь кликает по ссылке «рекомендованы» и **браузер отправляет запрос** на, допустим, /books/recommendations.
2. **Контроллер проверяет запрос**: пользователь должен быть залогинен. Или у нас должны быть подборки книг для незалогиненных пользователей. Затем контроллер обращается к модели и просит ее отдать список книг, рекомендованных пользователю N.
3. **Модель** обращается в базу данных, достает оттуда информацию о книгах: популярные сейчас книги, книги, купленные пользователем, книги, купленные его друзьями, книги из его wish list. На основе этих данных модель строит список из 10 рекомендованных книг и возвращает их контроллеру.
4. **Контроллер** получает список рекомендованных книг и смотрит на него. На этом этапе контроллер принимает решения! Если книг мало или список вообще пустой, то он запрашивает список книг для незалогиненного пользователя. Если сейчас идет акция, то контроллер может добавить в список акционные книги.
5. **Контроллер** определяется с тем, какую страницу показать пользователю. Это может быть страница с ошибкой, страница со списком книг, страница поздравление о том, что пользователь стал миллионным посетителем.
6. Сервер отдает клиенту страницу (**view**), выбранную контроллером. Она заполняется нужными данными (имя пользователя, список книг) и уходит к клиенту.
7. Клиент получает страницу и отображает ее пользователю.

В чем преимущества такого подхода?

Самое очевидное преимущество, которое мы получаем от использования концепции MVC — это четкое разделение логики представления (интерфейса пользователя) и логики приложения (серверная часть).

Второе преимущество — это разделение серверной части на две: умная модель (**исполнитель**) и контроллер (**центр принятия решений**).

В предыдущем примере был момент, когда контроллер мог получить от модели пустой список рекомендованных книг и решал, что ему с ним делать. Теоретически эту логику можно было бы засунуть сразу в модель.

Сначала при запросе рекомендованных книг модель бы решала, что делать, если список пустой. Затем пришлось бы в это же место добавить код, что делать, если сейчас идет акция, затем еще разные варианты.

Потом оказалось, что админ магазина хочет посмотреть, как будет выглядеть страница пользователя без акции, или наоборот сейчас акции нет, а он хочет посмотреть, как будет отображаться будущая акция. А методов-то для этого нет. Поэтому и было решено отделить центр принятия решений (контроллер) от бизнес-логики (модель).

Помимо изолирования видов от логики приложения, концепция MVC существенно уменьшает сложность больших приложений. Код получается гораздо более структурированным, и, тем самым, облегчается поддержка, тестирование и повторное использование решений.

Понимая концепцию MVC, ты как разработчик осознаешь, где нужно добавить сортировку списка книг:

- На уровне запроса к базе данных.
- На уровне бизнес-логики (модели).
- На уровне бизнес-логики (контроллер).
- В представлении — на стороне клиента.

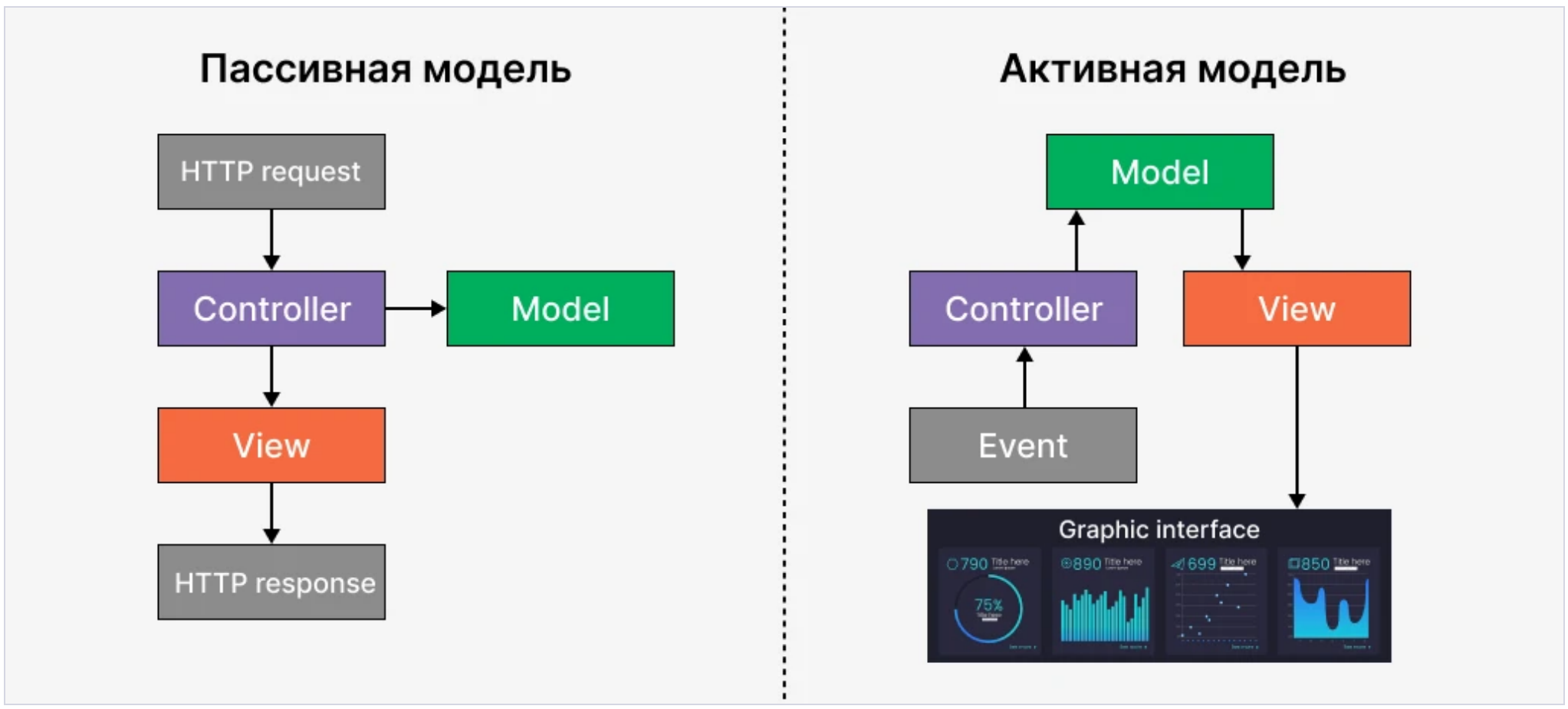
И это не риторический вопрос. Вот прямо сейчас и подумай: где и почему нужно добавить код по сортировке списка книг.

Классическая модель MVC

Взаимодействие между компонентами MVC реализуется по-разному в веб-приложениях и в мобильных приложениях. Это происходит из-за того, что веб-приложение — короткоживущее, обрабатывает один запрос пользователя и завершается, а мобильное приложение обрабатывает много запросов без перезапуска.

В веб-приложениях обычно используется "пассивная" модель, а в мобильных приложениях — "активная". Активная модель, в отличие от пассивной, позволяет подписываться и получать уведомления об изменении в ней. В случае с веб-приложениями этого не требуется.

Примерно вот так выглядит взаимодействие компонентов в различных моделях:



В мобильных приложениях (активная модель) активно используются **события** и механизм подписки на события. При таком подходе view (**вид**) подписывается на изменения модели. Затем, когда происходит какое-то событие (например, пользователь нажимает кнопку), вызывается **контроллер**. Он дает и **модели** команду на изменение данных.

Если какие-то данные изменились, то модель генерирует событие об изменении этих данных. Все view, которые подписались на это событие (для которых важно изменение именно этих данных), получают это событие и обновляют данные в своем интерфейсе.

В веб-приложениях все организовано немного по-другому. Основное техническое отличие — это то, что клиент не может получать сообщения со стороны сервера по инициативе сервера.

Поэтому контроллер в веб-приложении обычно не присылает view какие-либо сообщения, а отдает клиенту новую страницу, которая технически является новым view или даже новым клиентским приложением (если одна страница ничего не знает о другой).

В нынешнее время эта проблема частично решена с помощью таких подходов:

- Регулярный опрос сервера на счет изменения важных данных (раз в минуту или чаще).

- WebSocket'ы позволяют клиентку подписываться на сообщения сервера.
- Web-push-уведомления со стороны сервера.
- Протокол HTTP/2 позволяет серверу инициировать отправку сообщений клиенту.

‹ Предыдущая лекция

Следующая лекция ›

+19

Комментарии (8) популярные новые старые

JavaCoder

Введите текст комментария

Jh-007 Уровень 47 23 июня 2022, 16:10

Опечатка на иллюстрации: пассивная модель

Ответить +2

Сергей Уровень 39 22 июня 2022, 14:22

Немного путают рекомендации по засовыванию какой-либо логики в контроллер. В других местах говорят, что в контроллере должно быть минимум логики.

Ответить +3

Justinian Judge в **Mega City One** MASTER 9 августа 2022, 08:27

В контроллере не должно быть бизнес-логики.
Бизнес-логика это основная логика программы, например если программа по авиакомпании, то бизнес-логика это модели - Пилот, Кассир, Охранник, Техник, Инженер, это код отвечающий за начисление зарплаты, это некие условия - пилот не может быть старше 70 лет к примеру и тд.

Это все бизнес-логика, она должна быть в сервисе.

Контроллер более технический слой, его задача дёрнуть одно, потом второе, передать вызову третьему.

Контроллер это все равно часть программа, там все-равно будет код, и часть логики (технической) там все-равно будет, задача контроллера это обработка запросов и коммутация по сути.
Но иногда технически так получается, что контроллер владеет информацией о запросе, которая относится и к бизнес-логике, и ее очень морочно или нецелесообразно обрабатывать в другом месте, в 99% это будет информация о самом запросе/взаимодействии с пользователем, то есть в принципе относится к зоне ответственности контроллера.

Тогда, если этого кода немного, можно фрагментарно втиснуть в контроллер, не забывая главный принцип - не мы для паттернов, а паттерны для нас, но опять же, в рамки MBC это в принципе входит.

Просто нужно избегать классической ошибки начинающих - когда контроллеры становятся "толстыми" и вся или половина логики программы перемещается туда.

Валидация должна быть в сервисе, бизнес-логика - что и как преобразовывать должно быть в сервисе, другая логика, особенно объемная тоже в других сервисах или компонентах.

Контроллер отвечает за управление ходом программы, то есть передачу вызову Моделе или Вью, вызвать, получить ответ и отдать, контроллер отвечает за обработку запроса, техническую часть. Все остальное это уже другие компоненты.

Ответить +23

Oleg Khilko Уровень 51 15 августа 2022, 15:33

Justinian, ты пишешь: "Просто нужно избегать классической ошибки начинающих - когда контроллеры становятся "толстыми" и вся или половина логики программы перемещается туда."

Это достигается посредством инкапсулирования логики в других (тут какое правильно слово использовать? Объектах? Классах?)

Или какой-то другой подход применяется?

Ответить +2

Justinian Judge в **Mega City One** MASTER 16 августа 2022, 12:01

Для начала нужно осознать что такое контроллер.
Очень грубо на примере телевизора.
Что показывать и как - это внешний тюнер/флешка/какой-то чип. Это бизнес-логика, она отвечает за то, какой контент мы смотрим, собственно то, чего мы как потребитель хотим от девайса - видеть мультики, фильмы.
Это Модель. Это какие-то электронные компоненты внутри (тюнер, ИО с флешкой и тд)

Есть ЖК-панель, это View
Это то, на что мы смотрим, с чем взаимодействуем, особенно если она сенсорная, подходит, мы и смотрим на нее и взаимодействуем.

Есть Контроллер, это микросхема которая отвечает за обработку управляющих и обслуживающих сигналов, принял сигнал с пульта, переключил матрицу в режим 50Гц, подал яркость 70.

Здесь важно заметить, что Контроллер по сути связующее звено, он технически перенаправляет потоки, управляет работой ЖК панели и коммутирует сигнал с Модели , других микросхем.

То есть это более низкоуровневая логика которая касается именно УПРАВЛЕНИЯ флоу программы. Там не может быть логики "Если мультик для взрослых ТО детям не показывать", и тд, это бизнес-логика, это уже где-то в настройках должно быть.
Так и в нашей программе, если у нас есть программа, Животный питомец, в нем будет Модель, там будут классы СОбака, Кот, там будут классы описывающее как кормить котов, чем, как часто, это бизнес логика.

А в контроллере будет только связывание вводв/вывода от пользователя и вызов методов сервиса/Модели:
Сервис/Модель:
  Собака, Кот, Рыбка
  Как кормить животных
  Логика приюта

Контроллер:
View view;
Service service; //она же Модель

```
 switch (userInput)
  case giveFood:  service.giveFood();
  case showAllAnimals: view.show(service.getAllAnimals));
```

то есть роль Контроллера в том, что он связывает флоу и дергает тот или иной компонент, мы видим то сервис, то Вью.

Роль контроллера это обработка запросов юзера/компонентов и перенаправление действия другим компонентам.
Там не может быть логики в части какую рыбку и сколько кормить.

Ответить

+12

Justinian Judge в **Mega City One** MASTER — 16 августа 2022, 12:11

Это бизнес логика.

В контроллере логика более низкоуровневая, более чисто техническая, связанная с перенаправлением флоу программы.

Или другой пример Калькулятор:

View
Экран калькулятора
Кнопки

Model
```
public int sum(int a, int b) {
  return a + b;
}
```
...и так еще три операции или другие математические

Controller
Model model;
View view;

```
if (userCommand == SUM) {
  int result = model.sum(view.inputFromUser(), view.inputFromUser());
  view.showResult(result);
}
```

в чем у нас получается соль.
Если мы консольное приложение захотим сделать Спринговым, веб приложением или другим - мы меняем контроллер на Спринговый, МОДЕЛЬ ОСТАЕТСЯ такой же как и была, и у нас все будет работать.
Почему? Исключительно благодаря тому, что мы правильно распределили логику - инкапсулировали логику в каждом из слоев. Если бы представить, что часть бизнес логики была бы в контроллере, это было бы невозможно, мы бы рефакторили все и это было значительно сложнее.

Или другой момент, мы View решили поменять, и не консольное приложение, а писать в файл и читать с файла, опять же, или на запросы в интернет, опять же, мы меняем один компонент View, а все остальное у нас остается так же, и все работает.

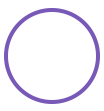
Самая распространенная ошибка в МВС консольных приложениях, это когда System.out.println размазывают по всему приложению, а чё нет? Где захотел, там и вывел, удобно! Но в программе на 2000 классов...заменять вывод в консоль на вывод в файл или запросы в сеть будет кошмаром.

А при правильной архитектуре, System.out.println будет 1 или максимум 2 и все они в одном месте, а вот уже вызываться будет 2000 раз. А потом когда мы захотим поменять на другое, мы меняем просто РЕАЛИЗАЦИЮ, а класс который будет вызывать - не будет знать, какая там реализация.

Суть этого паттерна в гибкости, ее сложно прочувствовать на небольшой программе в несколько классов, но просто представьте что у вас 2000 классов, и каждый из них смешивает все...рефакторинг будет практически невозможным.

Ответить

+16



Justinian Judge в **Mega City One** MASTER

16 августа 2022, 12:33

Поэтому логику и разбивают на слои, на пакеты, на классы, на методы, как в реальной жизни нам тоже гораздо удобней к примеру вести конспекты по предметам.

А мы как могли вести? Вот на среду расписание:
1. Английский
2. Сопромат
3. Социология

Мы могли бы как писать, вот начался учебный день, и мы в одну тетрадку все пишем по порядку, английский лекция за 16, сопромат лекция за 16, социология лекция за 16, и тд

Удобно.

А если потом повторять или готовиться к предмету? 5 страниц конспекта с этой тетрадки, потом поискать в другой...это неудобно. Поэтому мы все группируем за определенной логикой.

У начинающих же есть переломный момент, когда идет переход от написать код лишь бы сделать к архитектурным паттернам, слоям, и самое сложное - понять, почему так лучше, ведь на примере простых задач на несколько классов, когда все удобно в голове держать. Но нужно экстраполировать это на много тысяч классов и станет все на свои места.

Поэтому как конспекты, как мы одежду, вещи группируем, предметы, так и логику нужно группировать в зависимости от ее функционального назначения, чтобы легко было этими слоями манипулировать - читать, рефакторить, работать с ними.

MBC один из примеров такого подхода, есть много других + в жизни будет свой микс использоваться, но какой-то подход будет.

Джава это о backend, даже строя небольшой маленький проект мы применяем принципы архитектуры, которые применяются к большим проектам, как у даже самой маленькой модельке самолета будут крылья, нос, хвост и тд, поскольку общие принципы остаются те же.

Поэтому принцип простой.

Просто группируйте логику по функциональному, логическому или другому признаку.
одна строка которая делает одно, вторая другое, третья снова одно, четвертая третье, будет плохо поддерживаться, масштабироваться, а программы на джаве пишутся с расчетом на постоянный рефакторинг, масштабирование и поддержку.

Это не ассемблер или С, где раз написал, работает, перекрестился и забыл, пошел к след задаче.

Ответить

+17

Oleg Khilko Уровень 51

16 августа 2022, 16:48

Понял, большое спасибо за развернутый ответ!

Ответить

+5

ЯЗЫК ИНТЕРФЕЙСА

Русский

СКАЧИВАЙТЕ НАШИ ПРИЛОЖЕНИЯ