GeekBrains

Архитектура и шаблоны проектирования на Python

Паттерны веб-представления

© geekbrains.ru

На этом уроке

Паттерны веб-представления. WSGI-фреймворк. Шаблонизатор.

Оглавление

Паттерны веб-представления данных

Пользовательский интерфейс окна веб-браузера сейчас активно применяется в корпоративных приложениях и даже в однопользовательских системах. Причин тому несколько:

1. Привычный пользователю интерфейс.
2. Кроссплатформенность.
3. Стандартизованность и открытость стандартов.
4. Невысокие аппаратные требования.

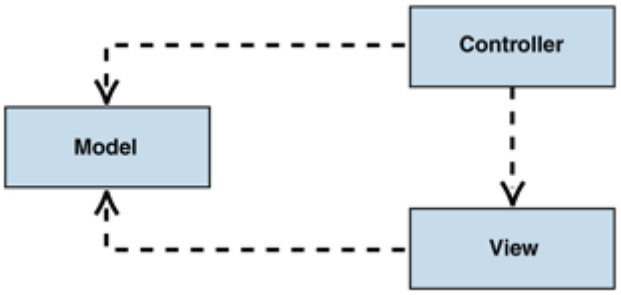
Именно поэтому важно изучать разработку UI-интерфейсов на основе проверенных веб-ориентированных решений.

MVC // Модель-Вид-Контроллер

Паттерн MVC — один из наиболее часто используемых. Он появился в конце 1970-х годов и использовался при разработке UI-фреймворка для платформы Smalltalk. С тех пор он играет важную роль в проектировании фреймворков пользовательского интерфейса и в понимании дизайна интерфейса как такового.

В основе MVC лежит идея об отделении представления информации от бизнес-логики (предметной области), моделирующей взгляд на объекты реального мира. Объекты, с которыми мы взаимодействуем в UI, более тесно связаны с платформой UI, нежели с логикой приложения. В то же время объекты бизнес-логики (далее будем использовать термин **модель**) должны быть полностью автономными и работать без привязки к UI, потому что:

1. Жизненный цикл UI явно не совпадает с жизненным циклом модели.
2. Если модель единственна, то представление UI может быть многовариантно. Например, тонкие и толстые клиенты, веб-интерфейс, мобильные приложения.
3. Тестирование модели, не зависящей от UI, значительно проще и легко поддаётся автоматизации на основе формальных критериев описания модели, чего нельзя сказать о тестировании UI.



- **Модель** — независимый компонент, поскольку она может существовать абсолютно автономно, в то время как **Контроллер** и **Представление** зависят от модели и друг от друга.
- Задача **Представления** — отобразить в UI необходимые поля модели.
- Внешние изменения, вносимые пользователем, обрабатываются **Контроллером**. Он модифицирует состояние модели и оповещает представление, что нужно обновить UI.

Поскольку MVC отделяет различные компоненты приложения, разработчики могут работать параллельно и одновременно над разными компонентами, не влияя друг на друга. Например, *Команда* может разделить разработчиков на frontend и backend. Backend-разработчики могут проектировать структуру данных и то, как пользователь взаимодействует с ней, не требуя завершения пользовательского интерфейса. Frontend-разработчики, наоборот, могут разрабатывать и тестировать пользовательский интерфейс приложения до того, как будет доступна структура данных.

Варианты

Пассивная модель используется, когда один контроллер манипулирует моделью исключительно. Контроллер модифицирует модель, а затем информирует, что она изменилась и должна быть

обновлена. Модель в этом сценарии полностью независима и от представления контроллера, а это значит, что у неё нет средств сообщать об изменениях в состоянии.

Пример пассивной модели — протокол HTTP (в чистом виде, без использования WebSocket). В браузере нет простого способа получать асинхронные обновления с сервера по инициативе сервера. Браузер отображает представление и отвечает на ввод пользователя, но самостоятельно не обнаруживает изменений в данных на сервере. Только когда пользователь явно производит какие-либо действия (ajax) или запрашивает обновление (refresh), сервер может отправить изменения браузеру.

Активная модель используется, когда модель изменяет состояние без участия контроллера. Это может произойти, когда другие компоненты, внешние по отношению к модели, меняют её данные, и изменения должны отражаться в представлениях. Поскольку только модель в этом контексте обнаруживает изменения в своём внутреннем состоянии, она должна уведомить представление, чтобы обновить отображение.

Тестируемость значительно повышается при использовании паттерна MVC. Тестирование компонентов затруднительно, когда они сильно взаимозависимы, особенно с компонентами пользовательского интерфейса. Когда возникает ошибка, трудно выделить проблему для конкретного компонента. Именно поэтому разделение ответственности — важная движущая сила MVC. Паттерн разделяет заботу о хранении, отображении и обновлении данных по трём компонентам, которые можно протестировать индивидуально. MVC не исключает необходимости в тестировании пользовательского интерфейса, но отделение модели от представления позволяет тестировать ее независимо и уменьшает количество тестовых сценариев для пользовательского интерфейса.

Преимущества

- Несколько разработчиков могут одновременно работать над моделью, контроллером и представлениями.
- Высокая связность — MVC позволяет логически группировать схожие действия на контроллере.
- Отделение бизнес-логики от контроллера.
- Простота модификации — из-за разделения обязанностей дальнейшее развитие или модификация становятся легче.
- Модели могут иметь несколько представлений: как форма GUI, как веб-страница или как REST-сервис.

Недостатки

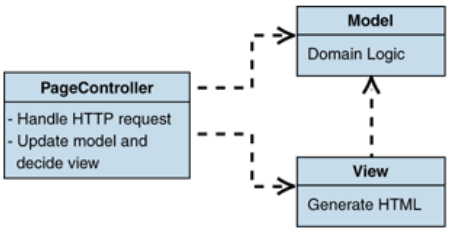
- Следование по структуре кода может быть осложнено, поскольку появляются новые уровни абстракции, что требует от пользователей адаптации и декомпозиции MVC.
- Разделение предмета разработки на три артефакта требует, чтобы разработчик(и) держал(и) в голове необходимость сохранять согласованность сразу нескольких компонентов.
- Разработчики, использующие MVC, должны быть квалифицированными в нескольких областях (Full Stack Developer).

Реализации

Самая известная и полная реализация паттерна MVC на Python — фреймворк Django. Остальные фреймворки в той или иной степени реализуют элементы MVC, но не полностью.

Page Controller // Контроллер страниц

Объект, который обрабатывает запрос к конкретной веб-странице или выполнение конкретного действия на веб-сайте.



Контроллер страницы получает её запрос, извлекает данные из него и других источников, вызывает обновление модели и перенаправляет запрос в представление. Представление, в свою очередь, зависит от модели, данные которой будут отображаться.

Определение отдельного контроллера страницы изолирует модель от специфики веб-запроса управлением сессией, разбором строки запроса или обработкой тела HTTP-запроса. В простейшем случае контроллер создаётся для каждой ссылки веб-приложения. Однако это может привести к значительному дублированию кода, поэтому обычно создаётся базовый класс BaseController для включения общих функций, а его наследники реализуют особенности обработки той или иной веб-страницы.

Преимущества

- Простота. Каждая веб-страница обрабатывается конкретным контроллером, который имеет дело только с ограниченной областью, и поэтому может оставаться простым.
- Плюсы повторного использования кода, благодаря концентрации общей функциональности в базовом классе BaseController.
- Разделение ответственности. Backend-разработчики могут сконцентрироваться на контроллере, в то время как frontend-разработчики — создавать и тестировать пользовательский интерфейс.

Недостатки

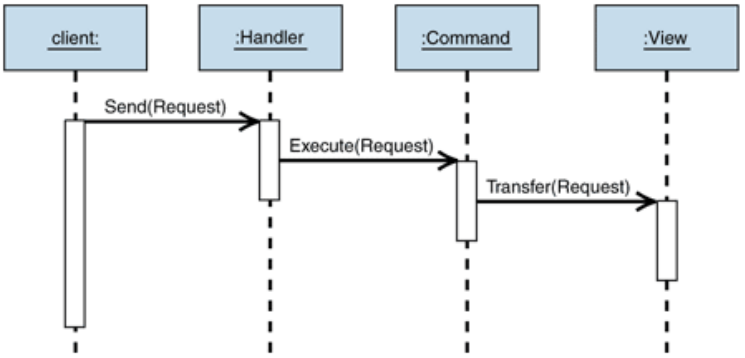
- Один контроллер на страницу. Ключевое ограничение для Page Controller — необходимость контроллера для каждой веб-страницы. Это хорошо работает для простых приложений, но терпит крах при значительном усложнении логики навигации в веб-приложении.
- Чрезмерное использование наследования, что может привести к негибким иерархиям.
- Зависимость от движка.

Реализации

Известны реализации на других языках программирования: Java Server Pages, ASPX и PHP. Для Python можно в качестве аналогии привести Zope Page Templates.

Front Controller // Контроллер запросов

Контроллер, который обрабатывает все запросы к веб-сайту.



Обработка запросов к веб-сайтам со сложной структурой подразумевает выполнение большого количества действий, общих для многих частей сайта: проверка безопасности, интернационализация и управление доступом к различным ресурсам сайта. Если не централизовать выполнение этих функций, это приведёт к дублированию большой части кода и усложнит поддержку консистентности управления.

Front Controller решает проблему децентрализации, направляя все запросы через один контроллер. Сам контроллер обычно реализуется как обработчик и иерархия команд. Для изменения поведения основного контроллера нередко применяется каскад декораторов.

В обязанности обработчика входит:

- Разобрать параметры запроса.
- Выбрать соответствующую команду на основе адреса запроса и его разобранных параметров.
- Передать управление выбранной команде для обработки.

Сами команды — также часть контроллера. Они представляют конкретные действия, описанные в шаблоне Command. Представление команд как отдельных объектов позволяет контроллеру взаимодействовать со всеми командами через общий интерфейс, в отличие от вызова определённых методов в общем классе команд. После того, как объект *Команды* завершит действие, *Команда* выбирает, какое представление использовать для отображения страницы.

Преимущества

- Front Controller координирует все запросы к веб-приложению. Единственный контроллер находится в идеальном месте для применения политики безопасности или управления доступом для всего приложения.
- Поскольку каждый запрос включает создание нового объекта Команды, объектам не нужно быть потокобезопасными (но это не отменяет требования потокобезопасности для остального кода).
- Конфигурируемость. На веб-сервере необходимо настроить только один Front Controller, что упрощает настройку. Использование динамических команд позволяет легко добавлять новые.

Недостатки

- Front Controller — единственное место, через которое проходят все запросы. Поэтому следует уделить особое внимание его дизайну с точки зрения производительности. Если для принятия

решения обработчик должен, например, выполнить запрос к базе данных или другую ненормированную по времени операцию, это будет плохой практикой.

Реализации

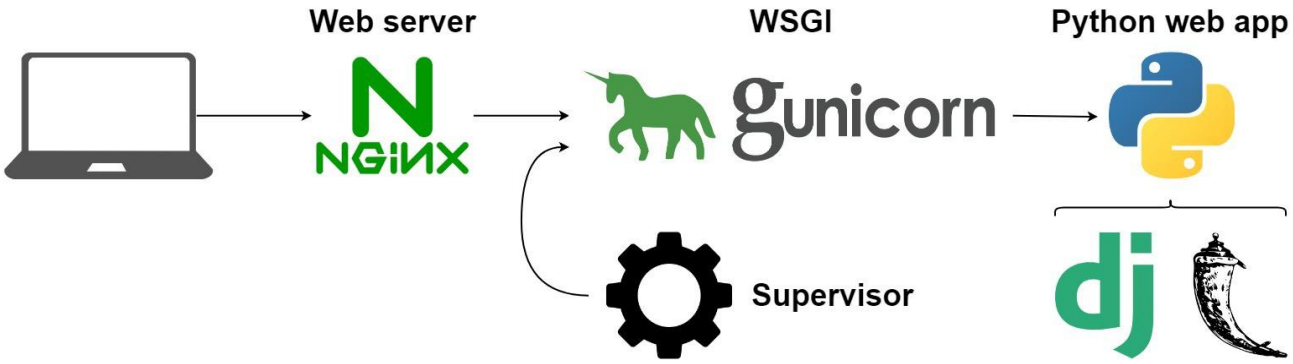
Пример реализации Front Controller — URL dispatcher, работающий в составе Django.

WSGI-фреймворк

В курсе мы реализуем свой WSGI-фреймворк. В нём мы сами реализуем паттерны Front Controller и Page Controller для их глубокого изучения и понимания, как устроены современные фреймворки.

Для начала разберём принцип работы веб-приложений на Python. Обычно приложение, работающее в production, состоит из следующих частей:

1. Балансировщик нагрузки (например Nginx, Apache, …). Его задача заключается в обработке пользовательских запросов и их перенаправления, а также в раздаче файлов static и media.
2. WSGI-сервер (например gunicorn, uwsgi). Его задача состоит в том, чтобы получать запросы от балансировщика и переводить их в вид, удобный для следующей обработки, а также предоставлять интерфейс для ответов.
3. WSGI-фреймворк (например Django, Flask, …). Содержит в себе обработку запросов и ответов и позволяет Python-разработчику создавать веб-приложения.



Рассмотрим пример обработки запросов, которую можно будет запустить с помощью uwsgi и gunicorn.

```
def application(environ, start_response):
    """
    :param environ: словарь данных от сервера
    :param start_response: функция для ответа серверу
    """
    # сначала в функцию start_response передаем код ответа и заголовки
    start_response('200 OK', [('Content-Type', 'text/html')])
    # возвращаем тело ответа в виде списка из bite
    return [b'Hello world from a simple WSGI application!']
```

Для запуска можно использовать gunicorn, uwsgi или их аналоги.

gunicorn

Ссылка: https://gunicorn.org/.

Установка:

pip install gunicorn

Запуск:

gunicorn simple_wsgi:application

uwsgi

Ссылка: https://uwsgi-docs.readthedocs.io/en/latest/WSGIquickstart.html

Установка:

pip install uwsgi

Запуск:

uwsgi --http :8000 --wsgi-file simple_wsgi.py

Объектом для запуска может быть любой callable-объект, т. е., либо функция, как в примере, либо класс с методом __call__.

Вызываемый объект принимает два параметра.

1. *environ* — словарь с данными запроса от пользователя, в нём мы можем получить все необходимые данные для обработки запроса.
2. *start_response* — функция для отправки кода ответа и заголовков, нужна для передачи заголовков ответа.

На выходе WSGI-сервер ожидает получить:

1. Код ответа и заголовки, переданные в функцию *start_response*.
2. Тело ответа после return в виде списка из набора байт.

На занятии мы реализуем базовую часть нашего WSGI-фреймворка, которая будет включать в себя обработку разных запросов от пользователя и паттерны Page Controller и Front Controller.

Использование шаблонизатора

Из примера видно, что для большой веб-страницы будет неудобно прописывать в коде весь её текст. Удобнее будет читать код страницы из текстового файла. Однако в таком случае возникает проблема передачи данных на страницу и возникает необходимость использования шаблонизатора.

Один из популярных шаблонизаторов — Jinja2.

Он позволяет не только читать шаблоны из файла и передавать в них данные, но и использовать специальные теги, а также базовые и включённые шаблоны.

```
"""
Используем шаблонизатор Jinja2
"""
from jinja2 import Template


def render(template_name, **kwargs):
    """
    Минимальный пример работы с шаблонизатором
    :param template_name: имя шаблона
    :param kwargs: параметры для передачи в шаблон
    :return:
    """
    # Открываем шаблон по имени
    with open(template_name, encoding='utf-8') as f:
        # Читаем
        template = Template(f.read())
    # рендерим шаблон с параметрами
    return template.render(**kwargs)


if __name__ == '__main__':
    # Пример использования
    output_test = render('authors.html', object_list=[{'name': 'Leo'}, {'name':
'Kate'}])
    print(output_test)
```

В примере показан простой способ чтения шаблона из файла в текущей папке и рендеринга шаблона с передачей в него параметров

Практическое задание

В этой самостоятельной работе тренируем умения:

1. Использовать паттерны page controller, front controller.
2. Использовать шаблонизатор.

Смысл:

Понимать и применять паттерны page и front controllers, понимать как устроены и работают WSGI-фреймворки. Использовать шаблонизаторы

Последовательность действий:

1. Создать репозиторий для нового проекта (gitlab, github, ...).
2. С помощью uwsgi или gunicorn запустить пример simple_wsgi.py, проверить, что он работает (*Эти библиотеки работают на linux-системах, документацию по ним можно найти в дополнительных материалах*).

3. Написать свой WSGI-фреймворк, используя паттерны Page Controller и Front Controller. Описание работы фреймворка:
    a. возможность отвечать на get-запросы пользователя (код ответа + html-страница);
    b. для разных url-адресов отвечать разными страницами;
    c. page controller — возможность без изменения фреймворка добавить view для обработки нового адреса;
    d. front controller — возможность без изменения фреймворка менять обработку всех запросов.
4. Реализовать рендеринг страниц с помощью шаблонизатора Jinja2. Документацию по этой библиотеке можно найти в дополнительных материалах.
5. Добавить любый полезный функционал в фреймворк, например обработку наличия (отсутствия) слеша в конце адреса, и так далее.
6. Добавить для демонстрации две любые разные страницы (например, главная и about, или любые другие).
7. Сдать задание в виде ссылки на репозиторий.
8. В readme указать пример, как запустить фреймворк с помощью uwsgi и/или gunicorn.

Дополнительные материалы

1. Паттерны проектировния vs архитектурные паттерны? — Хабр Q&A.
2. Описания паттернов проектирования. Паттерны проектирования. Шаблоны проектирования на Design pattern ru.
3. MVC vs. OOP.
4. Are Microservices the Future?
5. Jinja2 · PyPI.
6. Quickstart for Python/WSGI applications.
7. Gunicorn: Python WSGI HTTP Server for UNIX.

Используемая литература

1. Фаулер М. Шаблоны корпоративных приложений. М.: Вильямс, 2016.
2. Обзор паттернов проектирования.