GeekBrains

Архитектура и шаблоны проектирования на Python

Архитектура Python-приложений

На этом уроке

Виды и стили архитектур приложений.

Оглавление

Понятие «Архитектура программного обеспечения»

Концепция архитектуры программного обеспечения зародилась в исследовательских работах Эдсгера Дейкстры и Дэвида Парнаса на рубеже 60–70-х годов прошлого века. Эти учёные выявили критическую важность структуры программной системы. В 1990-х годах потребность изучения архитектуры получила дополнительные стимулы в результате роста сложности программного обеспечения. Это привело к формированию учения об архитектуре как обособленной отрасли научной мысли. Знания об архитектуре ПП (программного продукта) — результат целенаправленного поиска решения возникающих при создании ПП задач.

Можно говорить об архитектуре как о совокупности трудно изменяемых подходов (архитектурных решений) к проектированию ПП или как о максимально общем виде (структуре) проектируемого решения, без деталей реализации.

Например, Мартин Фаулер даёт такое определение: «Обычно это согласие в вопросе идентификации главных компонентов системы и способов их взаимодействия, а также выбор таких решений, которые интерпретируются как основополагающие и не подлежащие изменению в будущем». Если позже оказывается, что нечто изменить легче, чем казалось вначале, это нечто исключается из архитектурной категории.

Пример из реальной жизни: кирпич, из которого построено здание, легко заменяем по аналогии с лёгкой сменой реализации интерфейса или движка СУБД, в то время как общий облик здания крайне трудно изменить. Очевидно, что спорткомплекс в виде небоскрёба едва ли будет удобно строить и эксплуатировать в дальнейшем.

Впервые аналогию между разработкой программного обеспечения и классической архитектурой провел Кристофер Александер. Опираясь на его мысль, можно составить следующую мотивирующую таблицу.

Традиция	**Программирование**	**Функция**	**Мотивация в единицу времени**
Архитектор	Архитектор	Задаёт архитектуру системы	$$$
Проектировщик	Developer (Senior)	Проектирует систему, используя крупные блоки — шаблоны	$$
Строитель	Coder / программист (Junior)	Реализует	$

Мораль: «Плох тот солдат, который не мечтает стать генералом» (А.В. Суворов).

Стандарт IEEE 1471 даёт формальное определение. **Архитектура** — это базовая организация системы, воплощённая в её компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы.

Как в обычной жизни, где никто не строит здание без архитектурного проекта, так и у нас: подходить к созданию ПП без архитектуры — крайне легкомысленный шаг, за исключением простых задач, которые можно реализовать без проекта.

Преимущества архитектуры как абстракции сложной системы

1. Архитектура — это база для анализа поведения системы до её реализации, что позволяет устранить узкие места ещё на этапе проектирования.
2. Позволяет повторно использовать найденные ранее архитектурные решения, что, в свою очередь:
    a. Повышает скорость разработки.

    b. Повышает качество разработки.

    c. Снижает риски и вероятность провала.

    d. Снижает стоимость разработки.

    e. Увеличивает совместимость различных систем.

    f. Снимает необходимость снова и снова «изобретать велосипед».

3. Обеспечивает раннее принятие проектных решений, оказывающих в дальнейшем огромное влияние на разработку, внедрение и поддержку ПП, что позволяет:

    a. Выдерживать график разработки (внедрения, развёртывания и т. д.).

    b. Управлять стоимостью разработки.

    c. Распределить во времени принятие решений, «съесть слона по частям».

4. Облегчает взаимодействие разработчиков с кругом заинтересованных лиц (несведущих в программировании), чтобы создать систему, более полно отвечающую истинным требованиям заказчика.

Критерии хорошей архитектуры

С точки зрения современных разработчиков, список критериев хорошей архитектуры таков:

- Эффективность системы. В первую очередь программа должна решать поставленные задачи и хорошо выполнять свои функции в различных условиях.

- Гибкость системы. Любое приложение приходится менять со временем — меняются требования, добавляются новые. Чем быстрее и удобнее вносить изменения в существующий функционал, чем меньше проблем и ошибок это вызовет, тем гибче и конкурентоспособнее система.

- Расширяемость системы — возможность добавлять в неё новые сущности и функции, не нарушая основной структуры. Архитектура должна позволять легко наращивать дополнительный функционал по мере необходимости, причём так, чтобы внесение наиболее вероятных изменений требовало наименьших усилий.

- Тестируемость. Код, который легче тестировать, будет содержать меньше ошибок и надёжнее работать.

- Сопровождаемость. Хорошая архитектура должна давать возможность новым членам команды относительно легко и быстро разобраться в системе. Проект должен быть хорошо структурирован. По возможности в системе лучше применять стандартные, общепринятые решения, привычные для большинства программистов.

- Масштабируемость процесса разработки. Возможность сократить срок разработки за счёт добавления к проекту новых человеческих ресурсов.

В противовес критериям хорошо спроектированных архитектур используются и критерии «плохого дизайна», выдвинутые Сергеем Тепляковым:

- Его тяжело изменить, поскольку любое изменение влияет на слишком большое количество других частей системы (Жесткость, Rigidity).

- При внесении изменений неожиданно ломаются другие части системы (Хрупкость, Fragility).
- Код тяжело использовать повторно в другом приложении, поскольку его слишком трудно «выпутать» из текущего (Неподвижность, Immobility).

Формирование, выбор архитектуры. Декомпозиция

Используя принцип «от простого к сложному» (точнее, в данном контексте — «разделяй и властвуй»), необходимо разбить проектируемую систему на части, реализация которых вызывает меньше вопросов либо уже известна из накопленного опыта. Иными словами, нужно произвести декомпозицию системы.

При декомпозиции руководствуются следующими правилами:

1. Каждое разбиение системы порождает свой уровень сложности — иерархическая декомпозиция.
2. На всех уровнях иерархической декомпозиции система бьётся по какому-то одному признаку, например:
    a. Функциональная декомпозиция.
    b. Структурная декомпозиция.
    c. Временная декомпозиция и т. п.
3. Полученные подсистемы в сумме должны полностью описывать исходную систему, при этом не пересекаясь.

Вопросы проектирования архитектуры

При разработке архитектуры приложения необходимо найти ответы на следующие вопросы:

1. Какие части архитектуры фундаментальны?
2. Какие части архитектуры скорее всего будут изменяться в дальнейшем под натиском внешних факторов?
3. Проектирование каких частей архитектуры можно отложить?
4. Каковы основные варианты развития изменений и методы верификации предположений?
5. Что может привести к перепроектированию архитектуры?

Не пытайтесь создать слишком сложную архитектуру и не делайте предположений, которые не можете проверить. Лучше оставляйте свои варианты открытыми для изменения в будущем. Некоторые аспекты дизайна необходимо привести в порядок на ранних стадиях процесса, потому что их возможная переработка может потребовать существенных затрат. Такие области следует выявить как можно раньше и уделить им достаточное количество времени.

Основные принципы проектирования архитектуры

При проектировании архитектуры руководствуйтесь основными принципами:

1. Создавайте с расчётом на будущее. Продумайте, как со временем может понадобиться изменить приложение, чтобы оно отвечало вновь возникающим требованиям и задачам, и предусмотрите необходимую гибкость.
2. Создавайте модели для анализа и сокращения рисков. Используйте средства проектирования и системы моделирования, такие как унифицированный язык моделирования (Unified Modeling Language, UML), и средства визуализации, когда необходимо выявить требования, принять архитектурные и проектные решения и проанализировать их последствия. Тем не менее, не

создавайте слишком формализованную модель — она может ограничить возможности для выполнения итераций и адаптации дизайна.

3. Используйте модели и визуализации как средства общения при совместной работе. Для построения хорошей архитектуры критически важен эффективный обмен информацией о дизайне, принимаемых решениях и вносимых изменениях. Используйте модели, представления и другие способы визуализации архитектуры для эффективного обмена информацией и связи со всеми заинтересованными сторонами, а также для быстрого оповещения об изменениях в дизайне.
4. Выявляйте ключевые инженерные решения. В самом начале проекта уделите достаточно времени и внимания принятию правильных решений — это обеспечит создание более гибкого дизайна, куда можно вносить изменения без полной его переработки.
5. Рассмотрите возможность использования инкрементного и итеративного подхода при работе над архитектурой.
6. Начинайте с базовой архитектуры, правильно воссоздавая полную картину, после чего проработайте возможные варианты в ходе итеративного тестирования и доработки архитектуры. Не пытайтесь сделать всё сразу: проектируйте настолько, насколько это необходимо для начала тестирования вашего дизайна на соответствие требованиям и допущениям.
7. Усложняйте дизайн постепенно, в процессе многократных пересмотров, чтобы убедиться в правильности принятых крупных решений и лишь затем сосредотачиваться на деталях. Общая ошибка — быстрый переход к деталям при ошибочном представлении о правильности крупных решений из-за неверных допущений или неспособности эффективно оценить свою архитектуру.
8. При тестировании архитектуры дайте ответы на следующие вопросы:
    a. Какие допущения были сделаны в этой архитектуре?
    b. Каким явным или подразумеваемым требованиям отвечает архитектура?
    c. Каковы основные риски при использовании такого архитектурного решения?
    d. Каковы меры противодействия для снижения основных рисков?
    e. Улучшает ли эта архитектура базовую или это один из возможных вариантов архитектуры?

Эрозия архитектуры

Эрозия архитектуры выражается в разрыве между планируемой и фактически полученной архитектурой приложения при его реализации. Этот разрыв иногда называется «технический долг». Причиной его возникновения может быть как непродуманная архитектура, так и влияние внешних факторов. Например, давление бизнеса, когда заказчик требует реализации дополнительного функционала, для разработки которого зачастую нужно перепроектирование системы. В то же время и сами разработчики склонны к созданию предпосылок для возникновения «технического долга»:

1. Отсутствие тестирования — поощрение быстрой разработки и рискованных исправлений.
2. Использование временных, «уродливых», но быстро реализуемых решений для добавления нового функционала и исправления ошибок.
3. Отсутствие документации, что приводит к сложностям при необходимости доработки решения или внесения изменений спустя некоторое время.
4. Применение сильно связанных компонентов — оно приводит к отсутствию гибкости под натиском изменяющихся требований бизнеса.
5. Отсутствие взаимодействия, когда база знаний не распространяется по организации и страдает эффективность бизнеса, или младшие разработчики неправильно обучены наставниками.
6. Разрозненная разработка — может вызвать создание «технического долга» вследствие необходимости слияния изменений.

7. Откладывание важных изменений на потом. Как можно раньше внести важные и необходимые изменения гораздо дешевле последующего полного перепроектирования.

Восстановление архитектуры. Перепроектирование

Восстановление архитектуры — набор методов выделения архитектурной информации на основе анализа нижележащих слоёв дизайна, в том числе из программного кода.

Виды архитектуры приложений

Архитектура «клиент-сервер»

Клиент-серверная архитектура обычно разбивает приложение на две части по функциональному признаку. Клиентская часть создаёт запросы к серверу, сервер авторизует клиента, обрабатывает запрос и передаёт обратно ответом результат. Описание языка взаимодействия клиента и сервера называется **протоколом**. Типичные представители архитектуры «клиент-сервер»: WWW, FTP, электронная почта и т. п.

Основные преимущества архитектуры «клиент-сервер»:

1. Высокая безопасность. Все данные хранятся на сервере, который полностью контролирует доступ к ним (в теории).
2. Более простое администрирование вследствие централизованной авторизации пользователей сервером.
3. Простота обслуживания. Роли и ответственность вычислительной системы распределены между несколькими серверами, общающимися друг с другом по сети. Благодаря этому клиент гарантированно остается неосведомленным и не подверженным влиянию событий, происходящих с сервером (ремонт, обновление либо перемещение).

Тем не менее, у архитектуры клиент/сервер есть и недостатки:

1. Тенденция тесного связывания данных и бизнес-логики приложения на сервере, которое может негативно влиять на расширяемость и масштабируемость системы.
2. Зависимость от центрального сервера, что негативно сказывается на надёжности системы.

Многослойная архитектура

Многослойная архитектура описывает приложение как набор слоёв. Каждый слой реализует какую-то одну архитектурную задачу, используя при этом «сервисы» (в широком понимании этого слова) лежащего непосредственно под ним слоя, но не через слой ниже. В качестве примера многослойной архитектуры из смежных областей можно рассмотреть сетевую модель OSI:

	Единица данных	Уровень	Функция	Примеры протоколов
ОС	Поток	Прикладной	Прикладная задача	HTTP, SMTP, DNS, etc.
		Представления	Представление данных, шифрование, etc.	MIME, SSL
		Сеансовый	Взаимодействие хостов (на уровне ОС)	NetBIOS, именов. пайпы
	Сегмент	Транспортный	Соединение конец-в-конец, контроль передачи данных	TCP, UDP
Сеть	Пакет	Сетевой	Логическая адресация и маршрутизация пакетов	IP, ICMP
	Фрейм	Канальный	Физическая адресация	IEEE 802.3, ARP, DHCP
	Бит	Физический	Кодирование и передача данных по физическому каналу	IEEE 802.3

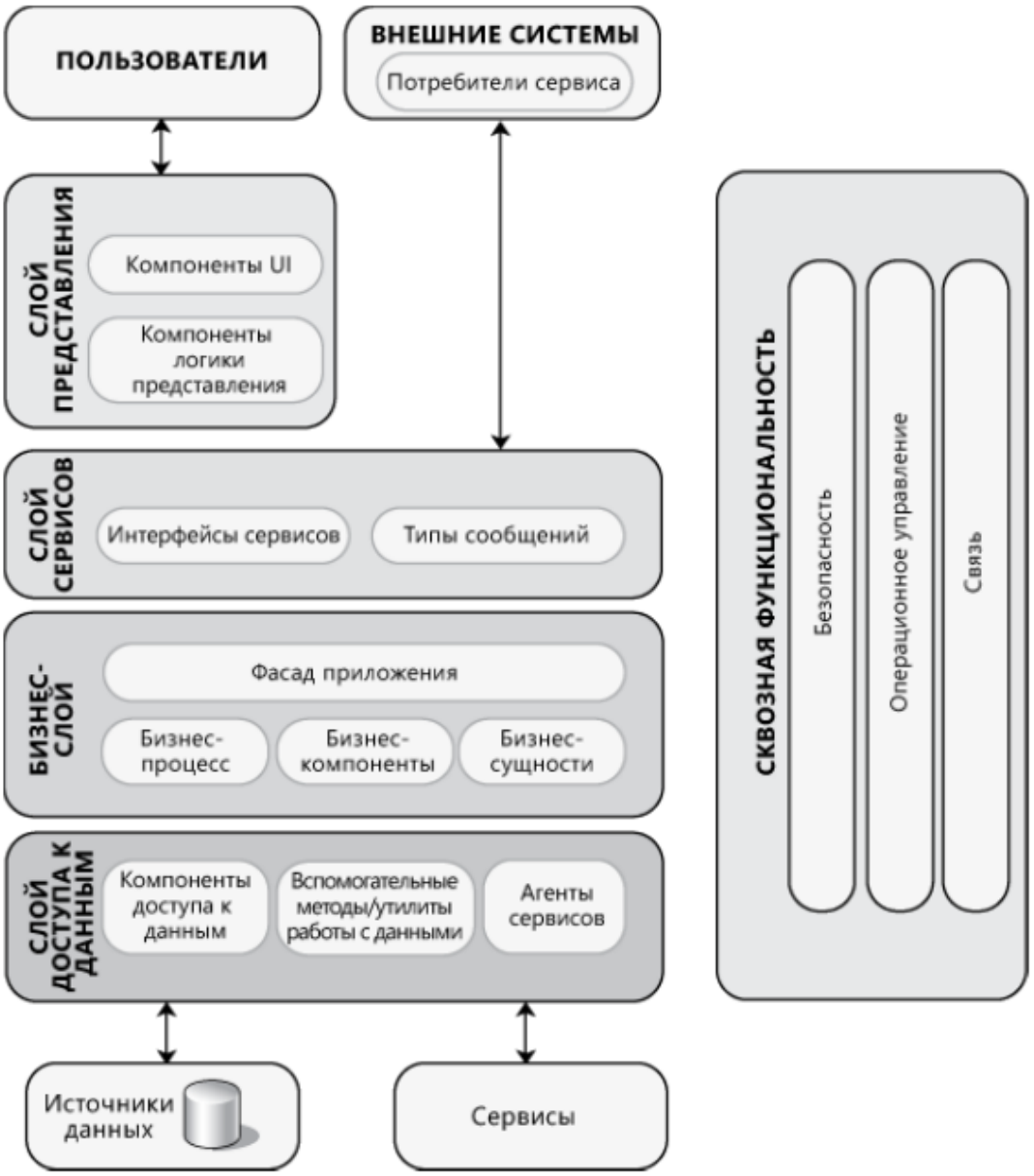
Источник: http://www.doctorrouter.ru/wp-content/uploads/2014/04/86ea4e.png

Преимущества расслоения:

1. Каждый слой — самодостаточный и индивидуально тестируемый.
2. Каждый слой независим от соседних слоёв.
3. Многовариантность реализации отдельно взятого слоя.
4. Стандартизация слоя позволяет многократно использовать его в разных приложениях.
5. Каждый слой может служить основой для нескольких различных слоёв более высокого уровня.

Недостатки:

1. Слои разбивают приложение по функциям, но не по смыслу. Добавление / модификация сущности, например поля БД, может спровоцировать каскад изменений в других слоях.
2. Избыточное расслоение может снизить производительность системы.



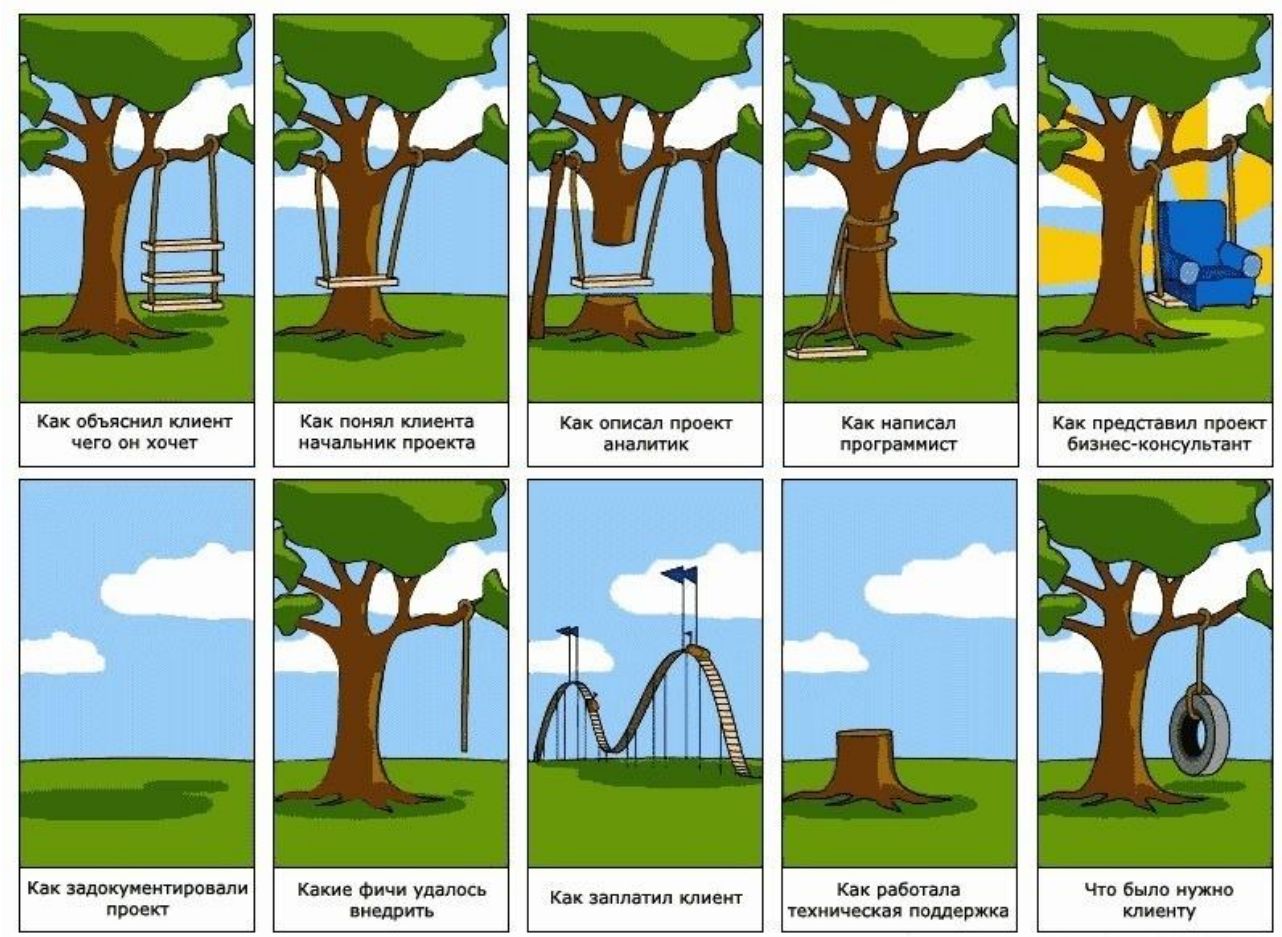
Архитектура типичного представителя многослойной архитектуры [12]

Проектирование на основе предметной области

Проектирование на основе предметной области (Domain Driven Design, DDD) — это объектно-ориентированный подход к построению архитектуры приложения с использованием понятий и языка предметной области. Этот язык разрабатывается на основе сущностей предметной области и в дальнейшем используется как разработчиками, так и экспертами области, не являющимися программистами, но владеющими всей полнотой знаний в этой сфере, например, экспертами со стороны заказчика.

Преимущества проектирования на основе предметной области:

1. Использование языка предметной области значительно упрощает обмен информацией.
2. Гибкая модель предметной области, написанная на языке предметной области, упрощает модификацию и развитие при изменении внешних условий.
3. Объекты модели предметной области прекрасно тестируются благодаря слабой связанности.



В целом DDD направлен на борьбу с широко известным примером (см. рис.) недопонимания между специалистами узкой направленности и заинтересованными лицами.

Сервисно-ориентированная Архитектура (SOA)

Сервисно-ориентированная архитектура разделяет функции на отдельные слабо связанные блоки (сервисы), которые могут располагаться как на одном узле сети, так и на разных. Сервисы имеют стандартизованные интерфейсы и могут обмениваться информацией, используя стандартные протоколы SOAP (Simple Object Access Protocol), REST, CORBA или Jini. Слабая связанность сервисов позволяет использовать в разработке отдельных из них разные стеки технологий. В рамках одного распределённого приложения могут успешно и взаимозаменяемо сосуществовать сервисы, созданные с использованием разных программных и аппаратных платформ, например, написанные на разных языках (Python, Java, C#, etc.). Каждый из сервисов внутри может использовать свою предметно-ориентированную архитектуру. Распределение функций приложения по слабо связанным небольшим блокам (микросервисам) позволяет очень быстро подключать и модифицировать различные функции системы.

Иногда при использовании архитектуры SOA сначала создают работающее монолитное приложение, а затем разделяют его на отдельные микросервисы. Такой подход получил название MonolithFirst.

Архитектура — шина сообщений (ESB)

Шина служб предприятия (ESB) реализует систему связи между взаимодействующими программными приложениями в сервис-ориентированной архитектуре (SOA). Основной принцип сервисной шины — концентрация обмена сообщениями между различными системами в единой точке, где при необходимости обеспечиваются транзакционный контроль, преобразование данных и сохранность

сообщений. Все настройки обработки и передачи сообщений также концентрируются в единой точке и формируются в терминах служб. Таким образом, при замене какой-либо информационной системы, подключенной к шине, не нужно перенастраивать остальные системы.

Компонентная архитектура

Компонентная архитектура описывает подход к проектированию и разработке систем с использованием методов проектирования программного обеспечения. Основное внимание в этом случае уделяется разложению дизайна на отдельные функциональные или логические компоненты, предоставляющие чётко определённые интерфейсы, содержащие методы, события и свойства. В данном случае обеспечивается более высокий уровень абстракции, чем при объектно-ориентированной разработке, и внимание не концентрируется на таких вопросах, как протоколы связи или общее состояние.

Создание WSGI-фреймворка. Запросы GET и POST. Получение данных из запросов

Из предыдущего занятия мы узнали, что данные запроса пользователя можно получить с помощью словаря environ, который приходит на вход нашего WSGI-фреймворка (callable-объекта)

В полноценном веб-приложении мы обычно работаем с двумя типами запросов (get, post) и у нас есть возможность получать данные запроса. Если это get-запрос, то данные передаются в адресной строке в следующем виде .../some_url/?name=max&age=18.

В случае с post-запросом данные приходят в теле запроса в виде набора байт

Таким образом, чтобы WSGI-фреймворк был универсальным, решим следующие задачи:

1) Как разделить запросы GET и POST.
2) Как получить данные из get-запроса (т. е., из адресной строки).
3) Как получить данные из post-запроса.

Для этого рассмотрим следующие примеры:

Как разделить запросы GET и POST

```
def application(environ, start_response):
    """
    :param environ: словарь данных от сервера
    :param start_response: функция для ответа серверу
    """
    # Метод которым отправили запрос
    method = environ['REQUEST_METHOD']
    print('method', method)
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [b'Hello world from a simple WSGI application!']
```

Как получить данные из get-запроса

```
def parse_input_data(data: str):
    result = {}
    if data:
        # делим параметры через &
        params = data.split('&')
        for item in params:
            # делим ключ и значение через =
            k, v = item.split('=')
            result[k] = v
    return result


def application(environ, start_response):
    """
    :param environ: словарь данных от сервера
    :param start_response: функция для ответа серверу
    """
    # получаем параметры запроса
    query_string = environ['QUERY_STRING']
    print(query_string)
    # превращаем параметры в словарь
    request_params = parse_input_data(query_string)
    print(request_params)
    start_response('200 OK', [('Content-Type', 'text/html')])

    return [b'Hello world from a simple WSGI application!']
```

Как получить данные из post-запроса

```
def parse_input_data(data: str):
    result = {}
    if data:
        # делим параметры через &
        params = data.split('&')
        for item in params:
            # делим ключ и значение через =
            k, v = item.split('=')
            result[k] = v
    return result


def get_wsgi_input_data(env) -> bytes:
    # получаем длину тела
    content_length_data = env.get('CONTENT_LENGTH')
    # приводим к int
    content_length = int(content_length_data) if content_length_data else 0
    # считываем данные, если они есть
    data = env['wsgi.input'].read(content_length) if content_length > 0 else b''
```

```
    return data


def parse_wsgi_input_data(data: bytes) -> dict:
    result = {}
    if data:
        # декодируем данные
        data_str = data.decode(encoding='utf-8')
        # собираем их в словарь
        result = parse_input_data(data_str)
    return result


def application(environ, start_response):
    """
    :param environ: словарь данных от сервера
    :param start_response: функция для ответа серверу
    """
    # получаем данные
    data = get_wsgi_input_data(environ)
    # превращаем данные в словарь
    data = parse_wsgi_input_data(data)
    print(data)
    start_response('200 OK', [('Content-Type', 'text/html')])

    return [b'Hello world from a simple WSGI application!']
```

Практическое задание

В этой самостоятельной работе тренируем умения:

1. Разделять get- и post-запрос внутри WSGI-фреймворка.
2. Получать и декодировать параметры post-запроса.

Зачем:

Чтобы уметь обрабатывать разные типы веб-запросов.

Последовательность действий:

1. Добавить в свой WSGI-фреймворк возможность обработки post-запроса.
2. Добавить в свой WSGI-фреймворк возможность получения данных из post-запроса.
3. Дополнительно можно добавить возможность получения данных из get-запроса.
4. В проект добавить страницу контактов на которой пользователь может отправить нам сообщение (пользователь вводит тему сообщения, его текст, свой email).
5. После отправки реализовать сохранение сообщения в файл, либо вывести сообщение в терминал (базу данных пока не используем).

Дополнительные материалы

1. https://medium.com/nuances-of-programming/%D0%BA%D1%80%D0%B0%D1%82%D0%BA%D0%B8%D0%B9-%D0%BE%D0%B1%D0%B7%D0%BE%D1%80-10-%D0%BF%D0%BE%D0%BF%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D1%8B%D1%85-%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%BD%D1%8B%D1%85-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%BE%D0%B2-%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9-81647be5c46f

Используемая литература

1. Архитектура системы.
2. Фаулер М. Шаблоны корпоративных приложений. М.: Вильямс, 2016. 544 с.
3. Александер К. Язык шаблонов. Города. Здания. Строительство. М.: Студия Артемия Лебедева, 2014. 1096 с.
4. Сетевая модель OSI.
5. Создание архитектуры программы или как проектировать табуретку.
6. Критерии плохого дизайна.
7. Что такое архитектура программного обеспечения?
8. ISO/IEC/IEEE 42010: Defining "architecture".
9. Немного про архитектуру | dev.by.
10. Архитектура приложения малой кровью / Хабр.
11. Концепция: Архитектура программного обеспечения.
12. Руководство Microsoft по проектированию архитектуры приложений.
13. Басс Л. и др. Архитектура программного обеспечения на практике. СПб.: Питер, 2006. 576 с.
14. Есть ли будущее за компонентной архитектурой? / Хабр.
15. Технический долг.
16. CAP theorem.
17. Brewer's CAP Theorem <= :julianbrowne.
18. Towards Robust Distributed Systems.
19. Эванс Э. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. М.: Вильямс, 2010. 448 с.
20. MonolithFirst.