# Введение

*СЛАЙД 1-2*

Сегодня на повестке дня фундаментальный материал по архитектуре приложений и конечно мы коснемся видов архитектуры приложений, чем они отличаются, какие есть плюсы и минусы.

*СЛАЙД 3*

Почему в нашей программе обучения эта тема? Она очень важна, ведь смысл обучения в Geekbrains – постараться выйти на уровень миддлов или хотя бы очень продвинутых джунов. Архитектурой занимаются именно миддлы, хотя могут и сеньоры, и архитекторы.

*СЛАЙД 4*

Рассмотрим возможные виды архитектур, где какая применяется, достоинства и недостатки.

И продолжим делать наш WSGI-фреймворк и будем разбираться с GET- и POST-запросами.

*СЛАЙД 5*

В курсе много абстракций, и паттерны будут абстрактные. Это не конструктор сайтов, где через строго определенные шаги мы собираем веб-приложение.

*СЛАЙД 6*

Идея в том, что, когда разрабатывается программная система, должен быть человек, следящий за этим. К разработке ПО понятие архитектура не очень подходит, но используется и даже существует такая должность – главный архитектор. Это очень престижная должность на Западе. Это человек, создающий каркас программы и следящий, за его поддержанием.

*СЛАЙД 7*

Об архитектуре много говорил Мартин Фаулер. Он много работал с бизнес-системами и впервые ввел понятие архитектуры.

Мартин дал такую формулировку – «.... 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».

Система растет, включает сложные компоненты и нужен человек, который будет и сможет за ней следить.

Итак, в системе существуют компоненты. Мы их определяем и дальше думаем, как они будут взаимодействовать в системе.

Очень важные слова здесь – «не подлежащие изменению в будущем».

Нам нужно найти такие компоненты и способы их взаимодействия, которые не будут менять в будущем. Именно они и сформируют архитектуру приложения.

Получается, архитектура – это скелет системы.

*СЛАЙД 8*

Джуниор – ему сказали, он выполняет и нужно проверить.

Миддл – ему сказали, он выполняет.

Сеньор – он берет часть системы, проектирует ее, и говорит джуниору или миддлу, какие задачи необходимо выполнить.

Архитектор – отвечает за всю систему (все ПО).

Из диаграммы следует, что основную работу выполняют миддлы и джуниоры.

А архитекторы почти ничего не пишут в плане кода, но получают больше.

*СЛАЙД 9*

Представим, что мы правильно определили архитектуру системы. Т.е. определили, что будет меняться в системе, а что нет.

Что мы получим?

1. *Повысится скорость разработки.* Мы уже будем знать, где конкретно что будет использоваться.
2. *Повысится качество разработки.*
3. *Снижение рисков и провалов.*
4. *Снижение стоимости разработки.*

Если не позаботиться об архитектуре, то эти положительные моменты уйдут в отрицательные.

*СЛАЙД 10*

Давайте посмотрим на таком упрощенном примере, чем отличается архитектура в классическом понимании (Строительство моста) от понимания в контексте Разработки ПО.

Строительство моста:

1. *Наличие проекта.* Когда мы строим мост, то уже все изначально знаем. Мы знаем бюджет предприятия, можем замерить ширину реки, мы знаем, какую массу должен держать мост.
2. *Обратная связь.* Права на ошибку у нас нет, а за жизнь можно построить сколько? Ну 1-10 мостов, вряд ли больше.

Строительство ПО:

Есть шутка, что если бы мы строили мост, как ПО, то получилась бы постройка короче на 3 метра и мы бы потом ее достраивали и в итоге получился бы вообще не мост, а что-то другое и оказалось бы, что этот мост нам совсем не нужен)).

1. *Неопределенность.* Не знаем, для чего система нужна и как будет работать. Сегодня можем делать одно, потом другое, третье и т.д. Поэтому здесь очень важно угадать, что именно меняться не будет.

Считается, что строительство города – ближе к разработке ПО.

Мы проектируем город, но не знаем, как там потом жители будут жить.

Мы просто делаем сектора – для жизни, промышленности и т.д.

1. *Метод грубой силы.* Мы можем совершить ошибку и при этом успешно ее исправить. И даже не единожды.

Делаем. Получаем ошибки, переделываем и т.д.

За жизнь можем поработать во многих больших и малых проектах.

*СЛАЙД 11*

Что нам предлагается сделать, чтобы разработать архитектуру системы?

1. *Создавайте с расчетом на будущее.* Обычно конкретных инструкций нет и нам приходится угадывать, опираясь на личный опыт. Угадать, что будет меняться в будущем, а что не будет. По моему опыту, будет меняться практически все, кроме бизнеса. Поэтому вскоре нам предложат использовать DDD. Что это, скоро узнаем. Поэтому предполагайте, что будет меняться практически все.
2. *Учитывайте вновь возникающие требования.* Это означает – решайте проблемы по мере их поступления.
3. *Используйте UML.* Визуализируйте все ваши задумки через графические представления. Это позволяет взглянуть на проблему с разных сторон.

*СЛАЙД 12*

1. *Используйте модели и визуализацию.* То же самое, что и пункт 3. Но отличие в том, что можно представить не в UML, а в каком-то другом формате.
2. *Используйте DDD.* Это очень важное понятие, его нужно запомнить.

Это разработка на базе домена (на основании предметной области).

*СЛАЙД 13*

Предметная область – это именно та задача, которую мы решаем.

Почему нам предлагается проектировать на ее основе?

Например, нам нужно сделать онлайн-магазин.

И предметная область здесь – это заказы, покупатели, товары, категории, корзина и т.д.

Например, мы делаем систему бух. учета.

И предметная область здесь – регламентный отчет, квартальный отчет, дебет, кредит, расчет зарплаты, сотрудники и все остальное, что относится к нашей задаче.

Например, мы делаем сайт по обучению йоге.

И предметная область здесь – поза, движение, тренер,

Т.Е это именно то, с чем мы работаем в проекте.

СМЫСЛ DDD – нам предлагают проектировать систему так, чтобы туда входили сущности, которые относятся именно к предметной области.

В Джанго – это модели.

В других проектах это может быть процедурное программирование – создание регламентного отчета, выдача ЗП сотрудникам и т.д. Это будут не классы, а функции, но тоже на основе предметной области.

Здесь есть очень важный момент.

Вернемся к формулировке Фаулера – “угадать не подлежащие изменению в будущем”

И вот эти наши компоненты выше – это единственное, что меняться не будет и это нам предлагается взять за основу.

Еще пример!

У нас есть фирма, занимающаяся перевозкой грузов. Раньше грузы возились на лошадях. А сейчас они возят на танкерах. Но задача фирмы от этого не изменилась. Они все также возят грузы.

Например, если ранее мы сделали класс Order, указав получателя, отправителя и т.д., то перейдя от лошадей к танкерам, ничего в контексте заказа не изменится.

Если бы мы проектировали не на основе DDD, например, написали скрипт, делающий что-то, нам бы пришлось его переписывать полностью.

*СЛАЙД 12*

1. *Выявляйте ключевые инженерные решения.*

Они вам пригодятся, как шаблоны.

1. *Усложняйте дизайн постепенно.*

Иначе что-то придется переделывать, а это лишняя работа.

*СЛАЙД 14*

Перейдем к понятию хорошей архитектуры, ведь нам нужно понять, что это и определить ее признаки.

1. **Эффективность**. Наша бизнес-система должна работать. Выполнять свою задачу.
2. Тестируемость. Это, во-первых, то, что система работает, т.е. эффективность, а во-вторых, то, что ее можно разделить на компоненты (модули) и протестить их работу. Менять не всю систему целиком, а только некоторую ее часть.
3. Стоимость. Чем меньше стоит, то вроде бы лучше. Но при сохранении эффективности.
4. Сложность. Если эффективна, то чем проще, тем лучше.
5. Универсальность. Бывает с гибкостью, этакий трансформер, но противоречит тогда сложности. Чем универсальнее, тем сложнее.
6. Целесообразность. Философский критерий. А нужна ли вообще такая система?
7. Понятная. Сопровождаемая и масштабируемая.
8. **Расширяемость**.
9. **Гибкость**.
10. Сопровождаемость. Можно подключить новых разработчиков.
11. Масштабируемость. Железо и разработчики. Накрутить серверов на систему. Кстати, WSGI это позволяет. Джанго-приложение можно раскидать по серверам и нагрузка будет по ним распределяться.

Разрабам нужно раскидать модули.

Давайте оценим применимость этих критериев к нашей задаче – созданию фреймворка.

Но есть важный момент – это не совсем бизнес-система. Это вспомогательная система для создания бизнес-системы (сайт).

- **модульная**. Разбита на компоненты.

- **сложность**. Должна быть по возможности простой.

- **понятность**. Хороший задокументированный код с правильным стилем имен переменных, ф-ций и т.д.

- **расширяемость**. У нас уже есть обработка GET и POST-запросов. А мы хотим еще сделать обработку POST-запросов (хотим сделать еще REST-фреймворк). Если мы это не вложили в систему, придется переписывать.

-**сопровождаемость**. Имеем возможность подключить разработчиков для самого фреймворка, для шаблонизатора, сам сайт будет делать третий и т.д.

-**создавайте с расчетом на будущее**. Может хотим добавить обработку json, текста, а не только html, как сейчас.

-**использовать uml**. Строим диаграмму системы.

- **использовать DDD**. Чем занимается фреймворк? Обработкой запросов отправкой ответов. Можно заранее заготовить набор методов, например, get\_request, front\_controllers, можно отдельный метод 404.

*СЛАЙД 15*

Перейдем к рассмотрению эрозии архитектуры.

Эрозия будет всегда и это необратимый процесс. Любая классная система устаревает!

Причины:

* Отсутствие тестирования
* Отсутствие документации
* Не модульность (сильно связанные компоненты)
* Отсутствие взаимодействия в команде
* Разрозненная разработка (когда сам что-то делаешь, не опираясь на ТЗ и т.д.)
* Не учитывание возможных изменений.

НО! Самое плохое – использование временных уродливых решений, хотя и быстрых (костылей).

Бизнес будет меняться быстро, и мы будем стараться внести быстро и изменения, но это будет уродливо. Поэтому эрозия неизбежна.

Поэтому, чем лучше применяются принципы, рассмотренные выше, тем эрозия происходит медленнее.

Чаще она происходит в жёстких системах, с хардкодом и т.д.

*СЛАЙД 16*

Можно ли восстановить архитектуру?

Это возможно, поговорим позднее!

*СЛАЙД 17*

Переходим к видам архитектур!

Часть вы знаете. Они фундаментальные, но мы рассмотрим и парочку новых!

*СЛАЙД 18*

**КЛИЕНТ-СЕРВЕРНАЯ**

Вернемся к формулировке Фаулера. Теперь поговорим о компонентах и особенностях их взаимодействия.

Самый простой пример – веб-сервер или сервер БД.

ПЛЮСЫ:

-высокая безопасность. Имеется в виду контроль, т.к. все лежит на сервере.

-простое администрирование.

-простота обслуживания. Все хранится на сервере, а нам лишь необходимо сделать интерфейс и пусть клиент подключается.

МИНУСЫ:

* Тенденция связывания данных и бизнес-приложений. Имеется в виду моделей и БД. Слава богу, что в Джанго за нас это делает ORM.
* Зависимость от центрального сервера. Упадет или будет блокирован Роскомнадзором.

*СЛАЙД 19*

**МНОГОСЛОЙНАЯ**

Классическая программа на Python, Java. Другое название – «монолит». Состоит из слоев.

ПЛЮСЫ:

-каждый слой независим

-его можно по-разному реализовать

-стандартизация.

Есть импорты, один слой использует другой и т.д.

МИНУСЫ:

-слои разбивают приложение по функциям, а не по смыслу. Нелогичная разбивка слоев.

-каскад изменения в слоях. Изменился один слой -> изменился и другой.

-чрезмерное расслоение может понизить производительность.

*СЛАЙД 20*

Но эти слои – как большая программа, написанная на Python.

Эта архитектура очень популярная, удобная, пока программа не перерастает во что-то огромное – в эдакий монолит.

*СЛАЙД 21*

**ПРОЕКТИРОВАНИЕ НА ОСНОВЕ ПРЕДМЕТНОЙ ОБЛАСТИ**

Это когда архитектура основана на слое Модели, и мы разрабатываем ее с учетом предметной области.

Определяем сущности и правила их взаимодействия.

ПЛЮСЫ:

-упрощает обмен информацией. Что в жизни, то и в программе.

-упрощает модификацию при изменении внешних условий. Например, мы сделали класс Покупатель. У него был метод «купить по кредитке», а стал «купить за наличку». Просто берем класс и добавляем новый метод.

-объекты предметной области прекрасно тестируются.

МИНУСЫ:

-иногда падает производительность

- плохо «ложится» на реляционные БД. Сами данные тяжело хранить. ORM это решает, но только отчасти.

*СЛАЙД 22*

**СЕРВИСНО-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА**

Сравним ее схематично с другими видами.

*СЛАЙД 23*

Монолитная программа. Пишется обычно на каком-то одном языке. Например, на Джанго. И обычно это одна БД.

SOA. Когда программа вырастает, мы делим ее на несколько частей. Но остается связь с одной БД.

Микросервисная. Делим на сервисы, плюс добавляем независимость от одной БД. Каждый сервис изолирован и у каждого своя БД. И сервисы маленького размера.

Как отличить вторую от третьей:

1. В третьей сервисы меньше.
2. Разные БД.

Зачем нужны 2-я и 3-я? Сделать системы более гибкими, когда сервис – одна минипрограмма. Сервисы в одной системе могут быть на разных языках.

ПЛЮСЫ:

-разделяет функции на отдельные слабо связанные блоки. По сути, каждый блок – это своя программа.

-стандартизованные интерфейсы. Речь об интерфейсах взаимодействия этих блоков. Т.е. они, например по сети общаются.

-мультиплатформенность. Быстрое изменение функционала. Каждая программа пишется на своем языке, поэтому ее можно быстро менять.

МИНУСЫ:

-требует инфраструктуры. Нужен архитектор, который будет за всем следить. Нужно много маленьких команд (на каждый сервис) и нужен DevOps-инженер (человек, который все правильно настроит).

*СЛАЙД 24*

Шина сообщений. Это уже развитие микросервисной архитектуры или SOA-архитектуры. Когда нужно подружить несколько мини-программ – «сервисов».

*СЛАЙД 25*

Мы можем «подружить» их через сетевые вызовы (через REST API), а можем через общую шину сообщений (брокер сообщений), например, RabbitMQ. Эти сервисы управляют шиной сообщений. И в которые складывают данные одни сервисы и забирают данные другие.

Переходим к примерам. Они несложные. В ДЗ их нужно будет «подружить» со своим фреймворком.

*СЛАЙД 28-29*

Но сначала о самом задании! Пока у нас были реализованы только GET-запросы без данных.

Нам нужно научиться разделять GET и POST запросы.

*СЛАЙД 30*

Последовательность действий:

0 – нужно понять, когда GET, а когда POST-запрос.

1 – добавить получение данных из POST-запроса, когда мы форму отправили.

2 – добавить получение данных из GET-запроса. Когда форму отправили GET-запросом или параметры передали в адресной строке.

…

cd /mnt/

cd c

cd Users/Дмитрий/Desktop

cd Архитектура\ и\ шаблоны\ проектирования\ на\ Python/

cd Урок\_2

cd Урок\ 2.\ Коды\ к\ уроку/

clear

Разделим запрос на GET и POST

**Листинг 1. get\_post.py**

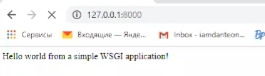
|  |
| --- |
| def application(environ, start\_response):  *"""  :param environ: словарь данных от сервера  :param start\_response: функция для ответа серверу  """  # Метод которым отправили запрос* method = environ[**'REQUEST\_METHOD'**]  print(**'method'**, method)  start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  return [b'Hello world from a simple WSGI application!']  *# gunicorn # pip install gunicorn # gunicorn get\_post:application  # uwsgi # pip install uwsgi # uwsgi --http :8000 --wsgi-file get\_post.py* |

У нас есть environ, в котором лежат данные о запросе и есть **'REQUEST\_METHOD'**.

Снова запускаем.

*uwsgi --http :8000 --wsgi-file get\_post.py*

Результат:





Переходим к get\_data. Что это?

**Листин 2. get\_data.py**

|  |
| --- |
| def parse\_input\_data(data):  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!']  *# gunicorn # pip install gunicorn # gunicorn get\_data:application  # uwsgi # pip install uwsgi # uwsgi --http :8000 --wsgi-file get\_data.py* |

Как передать данные в GET-запрос?

Например, так 

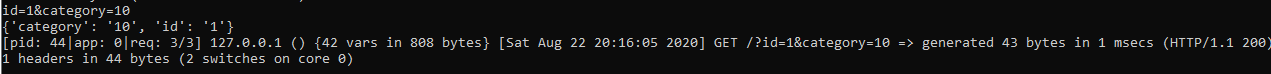
Снова запускаем.

*uwsgi --http :8000 --wsgi-file get\_data.py*

И вводим указанный выше запрос.

<http://127.0.0.1:8000?id=1&category=10>

Результат:



Нам нужно эти параметры оттуда достать и разделить!

Смотрим код. В коде у нас есть query\_string

Что это?



Теперь все это мы разделяем ф-цией parse\_input\_data

Теперь самый сложный пример

**Листинг 3. post\_data.py**

|  |
| --- |
| 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!']  *# gunicorn # pip install gunicorn # gunicorn post\_data:application  # uwsgi # pip install uwsgi # uwsgi --http :8000 --wsgi-file post\_data.py* |

Почти все то же самое, но данные в пост-запросе у нас идут в байтах и их нужно декодировать.

Приходит environ -> data = get\_wsgi\_input\_data(environ)

**CONTENT\_LENGTH** – длина тела данных

Функция get\_wsgi\_input\_data вернет нам данные (в байтах).

Считываем определенный кусок данных.

*# считываем данные если они есть*data = env[**'wsgi.input'**].read(content\_length) if content\_length > 0 else b''

Итак, мы взяли данные в байтах. Теперь берем parse\_wsgi\_input\_data и отправляем туда байты (тело пост-запроса).

Декодируем и они придут в таком же формате, что и в GET-запросе.

Т.е. после декодирования они будут в таком виде:



Берем уже знакомую функцию parse\_input\_data, она вернет словарь.

РЕЗЮМИРУЕМ:

Разделяем запросы на GET и POST.

Если у нас GET-запрос, то просто разделяем параметры через функцию parse\_input\_data

Если у нас POST-запрос, то понадобится эта же функция parse\_input\_data

чтобы поделить, еще одна ф-ция parse\_wsgi\_input\_data, чтобы декодировать ответ.

Ну а данные…они вот они

env.get(**'CONTENT\_LENGTH'**)

env[**'wsgi.input'**]

Вам нужно все это внедрить в свой фреймворк и сделать форму для обработки POST-запроса!