*Урок 2.* [*Архитектура*](https://gb.ru/lessons/192653) *Python-приложений*

Оглавление

[Введение 2](#_Toc89696594)

[Что такое архитектура 2](#_Toc89696595)

[Преимущества наличия архитектуры и шутка про мост 3](#_Toc89696596)

[Принципы разработки архитектуры 4](#_Toc89696597)

[Что такое хорошая архитектура 6](#_Toc89696598)

[Виды архитектур 6](#_Toc89696599)

[КЛИЕНТ-СЕРВЕРНАЯ 6](#_Toc89696600)

[МНОГОСЛОЙНАЯ 7](#_Toc89696601)

[ПРОЕКТИРОВАНИЕ НА ОСНОВЕ ПРЕДМЕТНОЙ ОБЛАСТИ 7](#_Toc89696602)

[СЕРВИСНО-ОРИЕНТИРОВАННАЯ 8](#_Toc89696603)

[ШИНА СООБЩЕНИЙ 9](#_Toc89696604)

[Дорабатываем фреймворк 9](#_Toc89696605)

[Разделяем GET- и POST-запросы 9](#_Toc89696606)

[Реализуем обработку GET- запросов с параметрами 10](#_Toc89696607)

[Реализуем обработку POST-запросов 11](#_Toc89696608)

# Введение

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

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.

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

И продолжим делать наш WSGI-фреймворк и научим его обрабатывать GET- и POST-запросы.

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

Идея в том, что, когда разрабатывается программное приложение, должен быть человек, следящий за всем процессом. 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 в исправном состоянии.

# Что такое архитектура

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.

Как же распределяются роли в проекте?

Джуниор – ему говорят, что делать, он выполняет, но нужно проверить результат.

Миддл – ему говорят, что делать, он выполняет и может сам себя проверить.

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

Архитектор – отвечает за всю систему (все ПО).

Основную работу по кодингу выполняют конечно миддлы и джуниоры.

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

# Преимущества наличия архитектуры и шутка про мост

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

Что мы получим?

* *Повышение скорости разработки.* Мы уже будем знать какие компоненты существуют в нашем проекте, как они связаны, будет меньше неопределенностей.
* *Повышение качества разработки.* При наличии четкой и понятной архитектуры проще поддерживать проект в работоспособном состоянии.
* *Снижение рисков и провалов.* Мы сможем оперативно вносить необходимые правки, устранять ошибки.
* *Снижение стоимости разработки.* Меньше работ по исправлению означает снижение временных, а значит и финансовых затрат.

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

Шутка про мост. Давайте посмотрим на упрощенном примере, чем отличается архитектура в классическом понимании («Строительство моста») от понимания в контексте разработки ПО.

Строительство моста:

Когда мы строим мост, то изначально имеем четкий и понятный проект. Мы знаем параметры реки и параметры будущего моста, знаем, какую массу должен выдерживать мост.

Теперь сама шутка:

Если бы мы строили мост, как ПО, то получилась бы постройка на 3 метра короче, 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)).

# Принципы разработки архитектуры

Что нам предлагается сделать, чтобы разработать архитектуру системы?

*Создавайте с расчетом на будущее.* Заранее все не предусмотреть, трудно сказать какие компоненты будут меняться в будущем, а какие останутся неизменными. Опыт показывает, что меняться будет практически все. Надо быть готовым к этому и разрабатывать архитектуру так, чтобы в будущем не пришлось выполнять глобальные переделки.

*Учитывайте вновь возникающие требования.* Это означает – решайте проблемы по мере их поступления. Заранее все не предусмотреть, значит не нужно пытаться.

*Используйте UML.* Визуализируйте все ваши задумки через графические представления. Это позволяет взглянуть на проблему с разных сторон.

*Используйте DDD.* Это очень важное понятие, его нужно запомнить.Это разработка на базе домена (на основании предметной области).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,

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.

*Выявляйте ключевые инженерные решения.*

Они вам пригодятся, как паттерны (шаблоны).

*Усложняйте дизайн постепенно.*

Иначе придется выполнять глобальные переделки, а это лишняя работа.

# Что такое хорошая архитектура

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

1. **Эффективность**. Наша бизнес-система должна в принципе работать, т.е. выполнять свою задачу.
2. **Сопровождаемость**. Мы должны иметь возможность подключать новых специалистов для обеспечения работоспособности системы.
3. **Расширяемость**. Должна быть возможность разработки и подключения модулей с дополнительной функциональностью.
4. **Тестируемость**. Должны быть возможность разделить систему на компоненты (модули) и протестировать их работу. Менять не всю систему целиком, а только некоторую ее часть.
5. **Понятность**. Хороший задокументированный код с правильным стилем имен переменных, функций и т.д.

# Виды архитектур

## КЛИЕНТ-СЕРВЕРНАЯ

Самый простой пример – веб-сервер или сервер БД.

ДОСТОИНСТВА:

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

Простое администрирование. Управление ведется централизованно, через центральный сервер.

Простота обслуживания. Все хранится на сервере, а нам лишь необходимо наличие интерфейса.

НЕДОСТАТКИ:

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 .

## МНОГОСЛОЙНАЯ

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

ДОСТОИНСТВА:

Каждый слой независим. Могут разрабатывать специалисты различных областей – бэкенд, фронтенд и т.д.

Каждый слой можно реализовать по-разному. Например, реализацию взаимодействия с базой данных можно реализовать с помощью различных ORM.

Стандартизация. Слои мы связываем с помощью импортов.

НЕДОСТАТКИ:

Слои разбивают приложение по функциям, а не по смыслу. Можно столкнуться с ситуацией нелогичной разбивки слоев.

Каскадность изменений в слоях. Изменения в одном слое потребуют изменений в других.

Чрезмерное расслоение может понизить производительность. Приложение будет «тормозить».

Как правило, когда мы делаем проект на Python, мы делаем монолит. Это было и на курсах Джанго, клиент-серверные приложения и т.д.

Эта архитектура очень популярная, удобная, пока программа не перерастает во что-то огромное. И тогда нужно задуматься о переходе к другому варианту архитектуры, например, сервисной или микросервисной.

## ПРОЕКТИРОВАНИЕ НА ОСНОВЕ ПРЕДМЕТНОЙ ОБЛАСТИ

В данном случае архитектура основана на слое Модели, и мы разрабатываем ее с учетом предметной области.

Определяем сущности и правила их взаимодействия.

ДОСТОИНСТВА:

Упрощает обмен информацией. Какие сущности существуют в жизни, такие мы реализуем в программе, используя ООП.

Упрощает модификацию при изменении внешних условий. Например, в нашем проекте существует 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. О важности написания тестов для своих проектов мы уже говорили.

НЕДОСТАТКИ:

Иногда может снижаться производительность. Но здесь дело скорее не в самой архитектуре, а в неправильной организации структуры компонентов приложения. Когда в нем излишнее расслоение.

Плохо «ложится» на реляционные БД. Сами данные тяжело хранить. ORM способствует решению этой проблемы.

## СЕРВИСНО-ОРИЕНТИРОВАННАЯ

Сравним ее схематично с другими видами.

Монолитная программа. Пишется обычно на каком-то одном языке. Например, на Джанго. И обычно связана с одной БД.

Сервисно-ориентированная (SOA). Когда программа вырастает, мы делим ее на несколько частей. Но остается связь с одной БД.

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

Как отличить от третьей:

1. В микросервисной сервисы меньше.
2. Каждый сервис связан со своей базой данных.

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

ДОСТОИНСТВА:

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, общаются по сети.

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

НЕДОСТАТКИ:

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).

## ШИНА СООБЩЕНИЙ

Шина сообщений. Это уже развитие микросервисной архитектуры или SOA-архитектуры. Когда нужно реализовать взаимодействие нескольких мини-программ – «сервисов».

Мы можем «подружить» их через сетевые вызовы (через REST API), а можем через общую шину сообщений (брокер сообщений), например, RabbitMQ. Когда в специальный канал складывают данные одни сервисы, а забирают данные другие.

# Дорабатываем фреймворк

Наш фреймворк уже умеет многое, но нужно научить его новому – понимать разницу между GET- и POST-запросом и научить фреймворк правильно их обрабатывать. Для этого внесем в исходный код фреймворка необходимые изменения.

## Разделяем GET- и POST-запросы

Информация о том, какой именно выполняется запрос, находится в словаре **environ**. Тип запроса скрывается за ключом **request\_method**.

**Листинг 1. Урок 2. Коды к уроку/get\_post.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   **def** application(environ, start\_response):  method = environ[**'REQUEST\_METHOD'**]  print(**'method'**, method) *# method GET* start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])  **return** [**b'Hello world from a simple WSGI application!'**]   **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |

После запуска перейдем на главную страницу и получим в терминале вот такой результат:

Теперь наш фреймворк «умеет» различать GET- и POST-запросы.

## Реализуем обработку GET- запросов с параметрами

С GET-запросом мы имеем дело, когда, например, открываем форму создания новой записи, например, нового пользователя, категории, товара и или открываем форму редактирования уже существующей записи.

Но GET-запрос умеет и передавать данные на сервер по аналогии с POST-запросом, когда эти данные передаются в качестве параметров в адресной строке:

Вот так может выглядеть GET-запрос с параметрами:

*127.0.0.1:8000?id=1&category=10*

Важно отметить, что при GET-запросе информация передается строкой прямо в URL.

Реализовать GET-запрос при открытии формы простая задача, гораздо сложнее сделать возможность отправки данных на сервер этим типом запроса.

**Листинг 2. Урок 2. Коды к уроку/get\_data.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   **def** parse\_input\_data(data):  result = {}  **if** data:   params = data.split(**'&'**)  **for** item **in** params:   k, v = item.split(**'='**)  result[k] = v  **return** result   **def** application(environ, start\_response):  query\_string = environ[**'QUERY\_STRING'**]  print(query\_string) *# -> 'id=1&category=10'* request\_params = parse\_input\_data(query\_string)  print(request\_params) *# -> {'id': '1', 'category': '10'}* start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])   **return** [**b'Hello world from a simple WSGI application!'**]   **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |

Мы получаем строку с переданными в GET-запросе параметрами и формируем из них словарь.

## Реализуем обработку POST-запросов

С POST-запросом мы имеем дело, когда сохраняем введенные в форму данные.

Важно отметить, что при POST-запросе данные передаются в теле запроса в байтовом формате.

**Листинг 3. Урок 2. Коды к уроку/post\_data.py**

|  |
| --- |
| **from** wsgiref.simple\_server **import** make\_server   **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, они приходит в строковом формате* content\_length\_data = env.get(**'CONTENT\_LENGTH'**)  *# приводим к int, если тело есть, иначе возвращаем 0* 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'**)  print(data\_str) *# 'id=1&category=10'  # собираем их в словарь* result = parse\_input\_data(data\_str)  **return** result   **def** application(environ, start\_response):  *# получаем данные* data = get\_wsgi\_input\_data(environ)  *# превращаем данные в словарь* data = parse\_wsgi\_input\_data(data)  print(data) *# -> {id: 1, category: 10}* start\_response(**'200 OK'**, [(**'Content-Type'**, **'text/html'**)])   **return** [**b'Hello world from a simple WSGI application!'**]   **with** make\_server(**''**, 8000, application) **as** httpd:  print(**"Serving on port 8000..."**)  httpd.serve\_forever() |