ChatGPT

Введение в серверную часть и языки программирования (Python)

Что такое Backend? – Аналогия с рестораном

Backend – это серверная часть приложения, невидимая пользователю. Если представить веб-приложение как ресторан, то фронтенд – это обеденный зал с меню и официантами, а бекенд – кухня и повара, которые готовят блюда по запросу [1]. Другими словами, всё, что происходит «за кулисами» сайта – обработка данных, бизнес-логика – относится к бекенду. Без исправно работающей «кухни» наш «ресторан» (сайт) не сможет обслуживать клиентов.

Принцип работы бекенда: приложение на сервере постоянно запущено и ожидает запросов от клиентов (например, от браузера пользователя). Когда приходит запрос, бекенд-код обрабатывает его: проверяет что нужно сделать, при необходимости обращается к базе данных, выполняет бизнес-логику (например, расчёты или проверку правил) и формирует ответ. Ответом может быть готовая HTML-страница (для отображения в браузере) или данные в формате JSON/XML для динамических приложений. После обработки сервер отправляет ответ обратно клиенту через интернет, и браузер пользователя отображает результат.

Пример JSON-ответа сервера: на иллюстрации показан фрагмент данных (список задач) в формате JSON, который бекенд может вернуть фронтенду по запросу (например, через API). Клиент (браузер или приложение) получает такие данные и использует их – например, отображает список дел пользователю.

Обратите внимание: конечному пользователю недоступен непосредственный доступ к серверной логике. Браузер посылает HTTP-запросы, а **бекенд** решает, что с ними делать – например, проверить данные формы, сохранить информацию, выполнить вычисления – и затем вернуть ответ. Именно благодаря бекенду сайт может реализовывать функциональность, требующую безопасного хранения данных или сложных вычислений (чего нельзя делать непосредственно на компьютере пользователя по соображениям безопасности или производительности).

Взаимодействие фронтенда с бекендом (схема)

Рассмотрим шаги взаимодействия между клиентом и сервером:

- **Клиент (браузер)** – отправляет запрос по определённому адресу (URL). Например, пользователь нажимает кнопку или переходит по ссылке на сайте. Это запускает HTTP-запрос из браузера.
- **Интернет** – запрос путешествует через сеть к серверу. Веб-браузер использует протокол HTTP для общения с сервером.
- **Веб-сервер** (nginx, Apache и др.) – получает запрос. В простом случае веб-сервер может сразу отдать статическую страницу (HTML/CSS/JS файлы). Но если нужен динамический контент или логика, веб-сервер передаёт запрос в серверное приложение (бекенд-приложение) для обработки. Например, nginx может проксировать запрос в наше приложение на Flask или Node.js.

- **Серверное приложение (бекенд)** – основной исполнительный механизм. Код на сервере (например, наш Python-/Node-/PHP-сервер) читает входящий запрос: определяет, что хочет пользователь (какой URL запрошен, какие данные переданы). Затем выполняет нужные действия: может обратиться к **базе данных** (например, запросить информацию о пользователе или сохранить новые данные) и обработать бизнес-логику согласно запросу.
- **Формирование ответа** – получив необходимые данные и выполнив логику, бекенд формирует ответ. Ответ может быть в виде HTML-разметки (если сервер генерирует готовую страницу) или в виде данных (например, JSON для Single Page Application или мобильного приложения). Бекенд отправляет этот ответ обратно через веб-сервер клиенту.
- **Ответ в браузере** – браузер получает ответ. Если это HTML-страница, он рендерит её (отображает пользователю интерфейс). Если это JSON (данные), обычно фронтенд-скрипт (JavaScript) на стороне клиента дальше обработает эти данные – обновит интерфейс, покажет уведомление и т.д. Цикл завершён: по запросу пользователя сервер выполнил работу и вернул результат.

Важно, что между клиентом и сервером взаимодействие происходит по протоколу HTTP. Существуют разные типы HTTP-запросов, но наиболее распространены **GET** и **POST**.
- **GET-запросы** используются для получения (чтения) данных. Они не изменяют состояние на сервере. Например, загрузка страницы или получение списка новостей – это GET. (В терминологии HTTP, GET – *идемпотентный* метод, то есть повторный запрос не изменит результат.) Проще говоря, GET предназначен «только посмотреть, не трогая» данные [2].
- **POST-запросы** применяются для отправки данных на сервер и возможного изменения состояния. Например, отправить форму с комментарием, загрузить файл, добавить новый пост – эти действия должны выполняться через POST. POST-запрос передаёт данные (в теле запроса) и обычно приводит к изменениям (создание, обновление чего-то на сервере) [2].

Кроме GET/POST есть и другие методы (PUT, DELETE, PATCH и др.), но в базовом вводном курсе достаточно понимать разницу между запросом на получение данных и запросом на отправку (изменение) данных.

Основные задачи бекенда и примеры работы

Бекенд часто называют «мозгом» приложения, который работает на удалённом сервере. Он отвечает за множество задач, необходимых для функционирования системы [3]:

- **Бизнес-логика:** выполнение правил приложения, обработка данных, различные вычисления. Например, проверка, что пользователь ввёл корректный пароль, или расчёт стоимости товара со скидкой – всё это делается на сервере.
- **Работа с базой данных:** сохранение данных, их поиск, обновление и удаление. Бекенд служит посредником между базой данных и клиентом – напрямую из браузера в базу данных запросы не отправляются, это небезопасно и неудобно. Сервер проверяет права доступа, формирует корректные запросы к БД (например, SQL-запросы) и обрабатывает полученные результаты.
- **API (Application Programming Interface):** бекенд предоставляет набор *эндпоинтов* (адресов URL), через которые фронтенд может получать данные или вызывать действия. По сути, API – это «контракт» между фронтом и бэком: какие запросы можно отправить и какой ответ ждать. Например, `GET /api/posts` может возвращать список постов в JSON.
- **Аутентификация и авторизация:** проверка личности пользователя и его прав. Бекенд получает, к примеру, токен или сессионный идентификатор из запроса и определяет, кто

это и что ему разрешено делать. Если у пользователя нет прав на определённое действие, бекенд вернёт ошибку (например, 401 Unauthorized).
- **Логирование, обработка ошибок, безопасность:** серверная часть также следит за корректной работой приложения – записывает ошибки, предотвращает подозрительные активности (например, слишком много запросов – может включить ограничение). Эти аспекты важны, но в нашем вводном обзоре мы глубоко в них не погружаемся.

Важно отметить, что веб-приложение обычно состоит из **двух частей, фронтенда и бекенда**, которые вынуждены взаимодействовать. Фронтенд отвечает за удобство и отображение, бекенд – за логику и хранение данных. Они обмениваются сообщениями (HTTP-запросами/ответами), чтобы вместе предоставить пользователю необходимый функционал [4].

Обзор серверных языков программирования

Серверную логику можно реализовать на разных языках. Выбор языка часто зависит от задач проекта, существующей экосистемы и навыков команды. Несмотря на разнообразие технологий, принципы работы серверов схожи – обработка запросов, манипуляция данными, взаимодействие с БД и т.д. Рассмотрим несколько популярных языков и сред для бекенд-разработки:

- **PHP.** Классический язык веб-разработки, исторически широко используемый для создания сайтов. На PHP построено огромное количество сайтов, в том числе популярнейшая CMS WordPress. (WordPress, работая на PHP, обслуживает ~43% всех сайтов в интернете! Это одна из причин, почему PHP до сих пор доминирует на серверной стороне [5].) PHP изначально создавался для генерации HTML-страниц на сервере – PHP-скрипты вставляются в код страниц и выполняются на сервере при каждом запросе. Современный PHP развивается: появились мощные фреймворки (Laravel, Symfony и др.), упрощающие создание приложений. PHP остаётся очень распространённым для веба [6], а порог входа в него невысокий, что когда-то поспособствовало его популярности.
- **Node.js (JavaScript).** Node.js позволяет использовать JavaScript – язык, изначально предназначенный для браузера – на серверной стороне. Это открыло возможность **full-stack** разработки на одном языке (JS) и быстро сделало Node.js популярным. Node.js отлично подходит для приложений реального времени и обмена сообщениями, благодаря событийно-ориентированной, неблокирующей архитектуре. Часто используется вместе с фреймворком Express для создания веб-серверов. Преимущество – фронтенд-разработчикам легко перейти к написанию бэкенда, не меняя язык. (На нашем курсе мы пишем фронт на JS, и благодаря Node можно использовать те же знания на сервере.)
- **Python.** Универсальный язык с простым, понятным синтаксисом. Очень популярный в самых разных областях – от веб-разработки до научных вычислений и машинного обучения. Для веба в Python есть множество фреймворков: **Django** и **Flask** – самые известные. Django – «полноценный» фреймворк со встроенными средствами для всего (от работы с БД до аутентификации), а Flask – более минималистичный, чтобы быстро поднять простой сервер. Python славится лаконичностью и читаемостью кода; например, в нём используется отступами для структурирования вместо фигурных скобок. В веб-разработке Python ценится за скорость разработки и богатую экосистему. (Также Python часто применяют в автоматизации, скриптах, тестировании – например, пишут авто-тесты для веб-приложений.) [7]
- **Java.** Язык, много лет являющийся корпоративным стандартом. Используется для крупных, нагруженных систем в банках, торговых площадках и прочих enterprise-приложениях. Отличается строгой статической типизацией, обширной экосистемой библиотек. В вебе Java наиболее известна благодаря технологиям **Spring Framework/Spring Boot**,

позволяющим создавать мощные веб-сервисы. Код на Java обычно компилируется в байт-код и работает на виртуальной машине (JVM), что даёт переносимость между разными операционными системами. Java-приложения часто обслуживают миллионы пользователей, будучи масштабируемыми и надёжными [8].

- **C# (.NET).** Язык от Microsoft, похожий по возможностям на Java (статически типизирован, высокопроизводителен). Используется в связке с платформой .NET. Веб-приложения на C# создаются с помощью **ASP.NET Core** – современного фреймворка, позволяющего писать как сайты (с рендерингом страниц), так и API для клиентов. C# особо популярен в среде Windows-разработки, в корпоративном секторе, а также для разработки игр (через движок Unity, хотя это уже не веб). Для веб-сервисов на C# также характерна высокая производительность и хорошая интеграция с инфраструктурой Windows-серверов.

Конечно, это не полный список. В бекенде используются и **Ruby** (с известным фреймворком Rails), и **Go** (от Google, набирает популярность благодаря простоте и эффективности), и многие другие. Каждый язык предлагает свои подходы, но повторимся – концептуально серверная часть на любом языке делает одно и то же: получает запрос, что-то вычисляет/сохраняет, и возвращает ответ. Поэтому, освоив принципы backend-разработки, в будущем вы сможете относительно легко переключаться между технологиями.

(Примечание: В рамках нашего курса мы не будем углубляться во все эти языки. Однако нужно знать об их существовании – в реальном мире команды выбирают разные инструменты. Мы же сфокусируемся на понимании общих принципов. Далее, для примера, рассмотрим язык Python, чтобы сравнить его с JavaScript.)

Краткий экскурс в Python – синтаксис и сравнение с JavaScript

Python – высокоуровневый язык программирования, известный простым синтаксисом. Давайте взглянем на небольшой код на Python и сравним его с тем, что вы уже знаете в JavaScript. Мы не ставим цель полностью обучить Python, но пару примеров помогут увидеть сходства и различия.

Пример 1: переменные и условный оператор
Вот фрагмент кода на JavaScript, который мы уже могли бы написать на фронтенде:

```
let name = "Alice";
let age = 21;
if (age >= 18) {
    console.log(`Привет, ${name}! Ты совершеннолетний.`);
} else {
    console.log(`Привет, ${name}! Тебе ещё нет 18.`);
}
```

Этот код объявляет переменную `name` и `age`, затем проверяет условие и выводит сообщение в консоль. Теперь тот же логический смысл на Python:

```
name = "Alice"
age = 21
if age >= 18:
```

```
    print(f"Привет, {name}! Ты совершеннолетний.")
else:
    print(f"Привет, {name}! Тебе ещё нет 18.")
```

Обратите внимание на отличия: - **Нет точки с запятой** в конце строк – в Python они не нужны (каждая новая строка считается концом инструкции автоматически). - **Блоки кода** (внутри `if`/`else`, функций, циклов) обозначаются **отступами**, а не фигурными скобками. В примере строки с `print(...)` начинаются с четырёх пробелов – это означает, что они внутри блока `if` или `else`. В JavaScript мы бы использовали `{ ... }` для того же. Если отступы поставить неправильно, Python не поймёт структуру кода. - Вместо `console.log` используется функция `print()`, которая выводит текст в стандартный вывод (консоль/терминал). - Строки можно форматировать с помощью `f"..."` – *f-строка* позволяет вставлять переменные внутрь фигурных скобок прямо в строке (похожее на шаблонные строки с `` `...` `` в JS). В примере внутри f-строки подставляется имя.

Если выполнить этот Python-код, при `age = 21` мы получим вывод:

```
Привет, Alice! Ты совершеннолетний.
```

Попробуйте заметить, что логика осталась абсолютно такой же, как и в варианте на JS – различается только «грамматика» языка. Python требует двоеточие `:` после условий и функций, использует ключевое слово `else` (как и JS), и в целом читается почти как псевдокод на английском.

Пример 2: цикл
Возьмём цикл, который вы уже видели в JavaScript. Например, вывести числа от 0 до 4:

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

На Python аналогичный цикл пишется так:

```
for i in range(5):
    print(i)
```

Функция `range(5)` генерирует последовательность чисел 0,1,2,3,4 (до 5 не включая). Цикл `for` пробегается по этой последовательности, и мы выводим `i`. Результат будет тем же – напечатаются числа 0,1,2,3,4 каждое с новой строки. Здесь опять видим отсутствие скобок и использование отступа для тела цикла.

Динамическая типизация: В JavaScript мы привыкли, что переменной можно присвоить значение любого типа, и тип может меняться (JS – динамически типизированный язык). Python тоже динамически типизирован: мы не объявляем явно тип переменной `name` или `age` – интерпретатор сам понимает, что `name` строка, а `age` число. Если потом `age = "двадцать один"` – Python это позволит (но тогда сравнение `age >= 18` вызовет ошибку, так как строку и

число сравнить нельзя). В отличие от Python и JS, такие языки как Java или C# требуют сразу указывать тип (int, string и т.д.), и не позволят менять его. Динамическая типизация упрощает начало работы, но требует быть внимательнее при работе с разными типами.

Общие элементы: переменные, условия, циклы, функции – имеются в любом языке программирования. Синтаксис и ключевые слова различаются, но, зная одну парадигму, освоить другую проще. Начинающим важно понять, что изучение второго языка становится легче: вы уже понимаете, что такое переменная или цикл, остаётся узнать, как это записать на новом языке. В нашем случае, зная основы JavaScript, вы уже способны прочитать и примерно понять простой код на Python. И наоборот, зная Python, легче затем разобраться в JavaScript – логика программирования едина.

(Мы рассмотрели Python здесь потому, что он прост для чтения и очень популярен. К тому же, он часто используется во фреймворках для веба. Но основной язык нашего курса – JavaScript/Node – тоже будет использоваться для серверной части ближе к проектам.)

Простейший сервер: пример кода на Python (Flask)

Чтобы понять, как код превращается в постоянно работающее серверное приложение, давайте создадим минимальный веб-сервер. Мы используем язык Python и микрофреймворк Flask для примера. Flask позволяет буквально в несколько строк описать обработку веб-запросов.

Предположим, мы установили Flask в нашу среду (через `pip install flask`). Создадим файл `server.py` со следующим содержимым:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello from server!"

if __name__ == "__main__":
    app.run()
```

Разберём, что здесь происходит:

- **Инициализация приложения:** мы импортировали класс `Flask` и создали объект `app`. Этот объект – наше веб-приложение/сервер. Ему передаётся имя модуля `__name__` для настроек. После этого приложение готово принимать настройки маршрутов.
- **Маршрут (route):** декоратор `@app.route("/")` связывает URL `"/"` (корневой адрес) с функцией `hello()`. То есть, когда придёт HTTP-запрос по адресу `/` (главная страница сервера), Flask вызовет функцию `hello`.
- **Обработчик запроса:** функция `hello()` просто возвращает строку `"Hello from server!"`. Flask автоматически возьмёт эту строку и сформирует HTTP-ответ, где тело ответа будет содержать эту строку (по умолчанию Flask считает это HTML-текстом).
- **Запуск сервера:** строка `app.run()` запускает встроенный веб-сервер Flask (для отладки). Приложение начнёт слушать входящие запросы на локальном сервере. По умолчанию

Flask запускается на адресе `http://127.0.0.1:5000` (127.0.0.1 означает «только на этом компьютере», порт 5000 – стандартный дефолтный порт Flask).

Теперь, **как всё проверить и увидеть в действии**: запускаем наш файл через Python интерпретатор. В терминале, находясь в папке с `server.py`, вводим:

```
$ python server.py
```

Flask выведет в консоль сообщение о запуске, например:

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Это значит, сервер запущен и ждёт запросов на порту 5000. Открываем браузер на компьютере, переходим по адресу **http://localhost:5000/** (localhost – то же, что 127.0.0.1). Браузер отправит GET-запрос на наш сервер, и… мы должны увидеть в окне браузера фразу: **“Hello from server!”** – именно ту строку, что возвращает функция. Мы своими глазами наблюдаем ответ, сгенерированный нашим бекенд-приложением! (Этот же результат можно получить, например, через Postman или curl, но браузер нагляднее). 9

Поздравляем, вы запустили простейший веб-сервер. Что же произошло под капотом? Запрос от браузера дошёл до нашего приложения, Flask сопоставил путь `/` с нужной функцией, выполнил `hello()`, получил строку и отдал её обратно. Веб-сервер обернул её в стандартный HTTP-ответ (добавил статус 200 OK, заголовки, тело с текстом) и отправил браузеру. Браузер получил ответ и отобразил текст на странице.

Обратите внимание: наш код **постоянно работает**, пока приложение запущено. Это не скрипт, который сразу завершился – это долговременно работающее приложение, «слушающее» порт 5000. Оно будет отвечать на каждый новый запрос, пока мы его не остановим (CTRL+C в терминале). Именно так работают серверы: они всегда в работе, ожидая новых запросов от пользователей.

Конечно, в реальности бекенд делает нечто более полезное, чем просто возвращать строку "Hello". Но даже такой минимальный пример демонстрирует принцип. Бекенд-разработчик обычно не пишет код низкого уровня для обработки сетевых соединений – за него эту работу делает сам фреймворк (Flask, Django, Express и т.д.) или веб-сервер (например, Apache может запустить PHP-файл автоматически при запросе). Разработчик описывает маршруты (какие URL обрабатывать) и логику в функциях, а библиотека берёт на себя всю рутину.

Для сравнения, аналогичный минимальный сервер можно написать и на **Node.js (JavaScript)**. Используя фреймворк Express, код выглядел бы так:

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
    res.send('Hello from server (Node)!');
});
app.listen(5000, () => {
```

```
    console.log('Server is running on http://localhost:5000');
});
```

Если запустить этот скрипт через Node (`node app.js`), он тоже будет непрерывно слушать порт 5000 и отвечать строкой `"Hello from server (Node)!"` на запросы к корневому URL. Принцип абсолютно аналогичен примеру с Flask/Python. Разница только в синтаксисе языка и некоторых деталях API. Понимая один пример, вы по сути понимаете и другой.

А как же **PHP**? Традиционно, PHP-приложения работают немного иначе: код PHP исполняется под управлением веб-сервера (Apache или nginx с PHP-FPM) при каждом запросе. Например, мы пишем файл `index.php`:

```
<?php
  echo "Hello from PHP!";
?>
```

Когда пользователь заходит на сайт, Apache сам запустит этот скрипт и вернёт его вывод. На время выполнения запросов PHP-скрипт живёт, потом завершается (в отличие от постоянно бегущего Flask/Node сервера). Однако результат для клиента тот же – на запрос пришёл ответ с нашим сообщением. Для запуска PHP-примера нужен установленный PHP-интерпретатор и сервер. В учебных целях можно воспользоваться встроенным сервером: выполнив `php -S localhost:8000` в папке с файлом, а затем зайти на `http://localhost:8000/index.php`. Вы увидите текст от PHP-скрипта. Таким образом, **PHP тоже позволяет реализовать серверную логику**, просто под другим "режимом работы". Современные PHP-фреймворки (Laravel и др.) абстрагируют эти детали, и разработчик так же пишет маршруты и функции контроллеров, не думая о низкоуровневом запуске.

Что произошло при запросе? (Разбор демонстрации)

Вернёмся к примеру с Flask-сервером и разберём шаги более детально, чтобы укрепить понимание: 1. **Запуск сервера:** Мы запустили наш Python-приложение, оно начало слушать порт 5000 на локальной машине. 2. **HTTP-запрос:** Когда в браузере мы открыли `http://localhost:5000/`, браузер сформировал HTTP GET запрос к этому адресу (по сути к `127.0.0.1:5000` с путём `/`). Этот запрос отправился через сетевой протокол к нашему приложению. 3. **Получение на сервере:** Flask-приложение приняло запрос. Внутри Flask произошёл матчинг URL – путь `/` соответствует декорированному маршруту, поэтому Flask вызывает связанную функцию `hello()`. 4. **Выполнение серверного кода:** Наш код внутри `hello()` исполнился. Он очень простой – просто вернул строку `"Hello from server!"`. В более реальных кейсах здесь могла быть работа с базой данных, проверка каких-то условий, логирование и др. – то есть вся та самая бизнес-логика. 5. **Формирование ответа:** Flask, получив результат от функции (строку), сформировал HTTP-ответ. По умолчанию Flask возвращает эту строку с HTTP статусом 200 (OK) и Content-Type 'text/html'. Если бы мы хотели вернуть, скажем, JSON, мы бы использовали `return jsonify(data)` или выставили соответствующий заголовок. 6. **Отправка ответа:** Flask отправил готовый HTTP-ответ обратно через сокет соединения к клиенту (наш браузер). Веб-сервер и транспорт TCP/IP позаботились о том, чтобы пакет с данными дошёл до браузера. 7. **Браузер отобразил ответ:** Получив ответ, браузер увидел, что это текст (HTML). В нашем случае просто текст "Hello from server!" – он и показался на странице. (Если бы это был JSON, браузер мог бы отобразить его как текст, либо, если это вызов через AJAX, фронтенд-скрипт бы обработал JSON и обновил страницу без перезагрузки.)

В этом цикле можно отметить, где могла подключиться база данных: на шаге 4 наш код **при необходимости обращается к БД**. Например, если это запрос `/profile`, сервер мог выполнить запрос к базе данных пользователей, достать информацию о текущем пользователе, и вставить её в ответ. Для клиента это прозрачно – он просто дожидается ответа чуть дольше, пока сервер общается с базой.

Также отметим, что весь обмен строится на HTTP – текстовом протоколе запрос/ответ. Мы пока не писали сами ни одного HTTP-заголовка вручную – за нас это сделал Flask/браузер. Но разработчик должен понимать концепцию: когда мы открываем страницу или отправляем форму, **всегда** идёт HTTP-запрос, и **где-то на сервере есть код, который его обрабатывает**.

Такое пошаговое разбирательство показывает: фронтенд и бекенд – как две половины разговора. Браузер спросил, сервер ответил. Попробуйте самостоятельно сформулировать, что делает сервер на каждом запросе – это и есть хлеб работы backend-разработчика.

Вопросы, которые могут возникнуть

– Насколько сложно создать полноценный сайт на бекенде?
Полноценный современный веб-сайт включает множество аспектов на сервере: обработка разных маршрутов (URL) в приложении, приём и валидация данных от пользователей, взаимодействие с базой данных (иногда не с одной, а с несколькими), авторизация пользователей, отправка уведомлений, интеграция с другими сервисами, обеспечение безопасности (например, хранение паролей в зашифрованном виде) и многое другое. Наш пример – крайне упрощённый сервер с одной функцией. Реальные бекенд-приложения могут содержать сотни маршрутов и тысячи строк кода логики. Тем не менее, принципы остаются теми же. Научиться писать простой бекенд может каждый; усложняется всё постепенно. Для начала важно понять концептуально: **что происходит после того, как пользователь нажал кнопку на сайте**. Если вы это уяснили, детали реализации – вопрос практики, опыта и изучения конкретных технологий.

– Что если я хочу сделать игру/мессенджер/другой тип приложения?
Веб-бекенд – лишь один из видов серверных программ. Есть серверы игровых приложений, чат-серверы, стриминговые серверы – они могут работать не по протоколу HTTP, а по специализированным протоколам или поверх WebSocket (для постоянного соединения с клиентом). Но базовая идея всё равно схожа: есть клиент (игра на компьютере или приложение на телефоне), и есть удалённый сервер, который обрабатывает запросы/сообщения от клиентов, выполняет логику (например, обновляет состояние игрового мира, пересылает сообщения собеседнику) и возвращает ответы. В нашем курсе мы фокусируемся на веб-разработке, поэтому говорим об HTTP. Понимая этот принцип, вы сможете потом разбираться и с другими архитектурами (хотя технических нюансов там будет больше).

– Нужно ли знать все эти языки (Python, PHP, Java и т.д.) чтобы быть бекенд-разработчиком?
Нет, профессиональные бекенд-разработчики, как правило, специализируются на одном стеке технологий, реже – на двух. Например, человек может быть Python-разработчиком и писать в основном на Django/Flask, или Java-разработчиком на Spring, или PHP-разработчиком на Laravel. Конечно, со временем можно изучить несколько языков – многие синтаксически похожи (C-подобные, как Java/JS/C#) или концептуально (Python и Ruby – скриптовые, динамические). Понимание разных языков расширяет кругозор и позволяет выбрать оптимальный инструмент под задачу. Но начинать надо с одного – в нашем курсе, напомним, бекенд-часть мы будем демонстрировать на **Node.js (JavaScript)**, чтобы вам было проще, ведь JS вы уже знаете с

фронтенда. Дополнительно мы показываем Python скрипты для сравнения, чтобы вы не думали, будто backend возможен **только** на JS.

– База данных – это тоже часть бекенда?
Да, база данных обычно относится к серверной части приложения. Часто выделяют даже три слоя: **Frontend** (клиентский интерфейс), **Backend** (серверное приложение, «бизнес-логика») и **Database** (хранилище данных). Бекенд взаимодействует с базой, но сама СУБД (система управления базами данных) – отдельный компонент. Веб-разработчик-бекенд должен уметь писать запросы к базе и проектировать схему данных, однако администрированием баз данных зачастую занимаются отдельные специалисты (DBA). В контексте нашего курса, мы познакомимся и с базами данных тоже – это важнейший элемент серверной части. Но это тема отдельного занятия. Главное сейчас понять: бекенд без базы часто бессмыслен, ведь нужно где-то хранить информацию (пользователей, посты, товары...). Поэтому почти любое серверное приложение работает в паре с БД.

Итоги и дальнейшие шаги

Мы рассмотрели, что скрывается за кулисами веб-приложения. **Backend** – это невидимый для пользователя механизм на сервере, который получает запросы от **Frontend** (видимой части сайта) и выполняет всю логику приложения. Бэк оформляет данные, взаимодействует с базами данных, применяет правила бизнеса и обеспечивает безопасность, а затем отправляет ответ обратно клиенту. Аналогия с рестораном помогает это представить: фронтенд – красивый зал и меню, бекенд – кухня, где готовится блюдо по заказу, а **API** – официант, который передаёт запросы и результаты между залом и кухней [10] [11].

На практике есть множество технологий для бекенда: вы узнали про несколько языков (PHP, JS/Node, Python, Java, C# и др.) и поняли, что выбор языка не меняет ключевых принципов работы серверной части. Мы даже запустили простой сервер, чтобы увидеть вживую, как фронтенд и бекенд обмениваются сообщениями.

В следующих занятиях мы углубимся в то, **как фронтенд общается с бекендом** более подробно – будем говорить про **API** (Application Programming Interface) и формат обмена данными (например, JSON). Также впереди знакомство с базами данных: рассмотрим, как данные хранятся и как запросы извлекают информацию. Все эти компоненты вместе образуют полноценную картину веб-разработки.

Теперь, когда вы нажимаете кнопку «Отправить» на сайте, вы представляете, какой путь проделывает ваш запрос и что делает сервер, прежде чем страница обновится. Это и есть цель нашего введения – дать общее понимание кухни веб-разработки. Впереди ещё много нового, но фундамент заложен: **Frontend + Backend = полный цикл работы веб-приложения**, и вы познакомились с ролью серверной части в этом дуэте.

---

[1] [3] [4] В чем разница между Backend и Frontend? — Хак Собесов
https://hacksobesov.com/questions/golang-developer/v-chem-raznicza-mezhdu-backend-i-frontend/

[2] forms - When should I use GET or POST method? What's the difference between them? - Stack Overflow
https://stackoverflow.com/questions/504947/when-should-i-use-get-or-post-method-whats-the-difference-between-them

5 PHP Usage in 2025 | Emmanuel Ojukwu
https://www.linkedin.com/posts/ifeanyi-ojukwu_usage-statistics-of-server-side-programming-activity-7328196100685791232-pDST

6 7 8 Frontend vs Backend разработка 2025: полное руководство для начинающих программистов
https://devpulse.uz/ru/blogs/frontend-va-backend-nima-farqlari-va-yonalishlari

9 Quickstart — Flask Documentation (3.1.x)
https://flask.palletsprojects.com/en/stable/quickstart/

10 11 Frontend vs Backend: Explained with a Restaurant Analogy | by Devadharshini Karthikeyan | Jul, 2025 | Medium
https://medium.com/@devadharshinik2012/frontend-vs-backend-explained-with-a-restaurant-analogy-b0e94900af2a