# Определения

Ресурсы – конечный файл лежащий на жестком диске хоста.

Localhost – 0.0.0.0 это localhost.

# URL/URI

URI (Uniform Resource Identifier)

URL (Uniform Resource Locator)

URL частный случай uri. URI – какой то идентификатор ресурса, а url это именно адрес, включающий метод поиска http или ещё другой и эндпоинт.

# JSON

JSON (JavaScript Object Notation) — это формат обмена данными, основанный на текстовом представлении структур.

Примеры валидных JSON-значений:

|  |  |
| --- | --- |
| **Тип данных** | **Пример JSON** |
| Строка | "hello" |
| Число | 123 |
| Логическое | true, false |
| Null | null |
| Массив | ["a", "b", "c"] |
| Объект | {"key": "value"} |

**ВАЖНО!**

string = ‘text’ - НЕ БУДЕТ распознан json форматом как текст, нужно ‘“text”’

# HTTP протокол

1. HTTP (HyperText Transfer Protocol) –

## Как это работает

1. Жёсткие стандарты (Договорённости)

Протокол HTTP — это именно что **договорённость**, описанная в технических спецификациях (RFC). Эта договорённость очень строгая и определяет:

* **Структуру запроса и ответа:** Любое HTTP-сообщение ДОЛЖНО иметь стартовую строку, заголовки и тело (которое может быть пустым).
  + **Стартовая строка:** Для запроса — METHOD URI HTTP/VERSION (например, GET /index.html HTTP/1.1). Для ответа — HTTP/VERSION STATUS\_CODE STATUS\_MESSAGE (например, HTTP/1.1 200 OK).
  + **Заголовки (Headers):** Пары Имя: Значение, которые несут мета-информацию (Host:, Content-Type:, User-Agent: и т.д.).
  + **Тело (Body):** Необязательная часть, отделённая от заголовков пустой строкой. Содержит данные (JSON, файл, форма и пр.).
* **Семантика методов:** GET — получить данные, POST — создать, PUT — обновить, DELETE — удалить. Это не enforced технически (сервер может удалять данные по GET-запросу), но это грубейшее нарушение конвенции, с которым никто не будет работать.
* **Коды состояний (Status Codes):** 200 — OK, 404 — Not Found, 500 — Internal Server Error и т.д. Это универсальный язык для понимания результата запроса.

**Если программа (клиент или сервер) не следует этим правилам, она не может участвовать в "разговоре" по HTTP.** Это как пытаться говорить по-английски, используя русскую грамматику — вас просто не поймут.

1. Автоматизация (Чёткие алгоритмы)

Поверх этих договорённостей работает мощная **автоматизация**.

На стороне клиента (Браузер, мобильное приложение, скрипт)

1. **Вы строите запрос:** Вы (разработчик) используете библиотеки (например, fetch в JS, requests в Python, OkHttp в Java), чтобы создать структуру запроса.
2. **Библиотека работает "по шаблону":** Вы говорите библиотеке: "Сделай POST-запрос на такой-то URL, положи эти данные в body и добавь вот эти заголовки".
3. **Библиотека-автомат:** Библиотека **автоматически** формирует валидное HTTP-сообщение **строго по стандарту**: правильно оформляет стартовую строку, заголовки, ставит пустую строку перед body, кодирует данные и т.д. Вам не нужно вручную собирать байты.
4. **Отправка:** Операционная система通过网络 (TCP/IP) отправляет этот сформированный пакет на сервер.

На стороне сервера (Nginx, Apache, Node.js, Django, Spring)

1. **Веб-сервер (Nginx/Apache) — первый автомат:** Он принимает "сырые" байты от сети.
2. **Парсинг:** Он **автоматически** парсит входящий поток байтов, распознаёт где заканчивается стартовая строка, где заголовки, где пустая строка, и где начинается body. Он делает это по тем самым правилам HTTP.
3. **Преобразование:** Веб-сервер преобразует распаршенные данные в удобную структуру (объект Request), которую он передаёт вашему приложению (например, на Python или Java).
4. **Ваше приложение (ваша логика):** Вы получаете уже готовый, разобранный запрос. Вы не парсите строки вручную. Вы просто смотрите на свойства объекта:

## Общее устройство запроса

1. HTTP сообщение – концептуально состоит из 3 частей:
2. Путь – адрес хоста в сети интернет + адрес ресурса на хосте

Для скорости ответа сразу в адресе ресурса могут содержаться переменные, помогающие найти ресурс

1. Хэдер – метаданные к боди.
2. Боди – некоторые данные.
3. С помощью специального алгоритма кодирования все эти три части компилируются в один сериализованный битовый запрос, который по интернету отправляется по адресу в *пути*.
4. Все эти три части HTTP могут быть сформированны совершенно независимо, главное чтобы в алгоритм потом передать правильно.

### Запрос и ответ

1. Однако, коммуникация http сообщениями также строго рекламентирована на запросы и ответы: Клиент отправляет запрос, а сервер возвращает ответ, далее коммуникации нет.

Запрос и ответ немного различаются элементами, но сохраняют общую структуру

1. **Установка соединения:** Клиент (например, браузер) устанавливает TCP-соединение с сервером.
2. **Отправка запроса:** Клиент отправляет **ровно одно** HTTP-сообщение-запрос.
3. **Обработка и ответ:** Сервер обрабатывает этот запрос, формирует и отправляет **ровно один** HTTP-сообщение-ответ.
4. **Разрыв соединения:** Соединение может быть закрыто (особенно в HTTP/1.0). В современных версиях (HTTP/1.1 с Keep-Alive, HTTP/2, HTTP/3) соединение может оставаться открытым для последующих запросов, но **логика «один запрос — один ответ» не меняется**.

1. Структура запроса (request / query):

|  |  |
| --- | --- |
| POST /api/users?limit=10&page=1 HTTP/1.1 | Path  (Path, query parameters)  METHOD URI HTTP/VERSION |
| Host: example.com  Content-Type: application/json  Authorization: Bearer abc123def456  User-Agent: Mozilla/5.0 | Header  (Header parameters)  keys: value |
|  | Пустая строка отделяющая header от body |
| {  "name": "John Doe",  "email": "john@example.com",  "age": 30,  "isActive": true  } | Body  Любой формат данных (может быть пустым) |

1. Структура ответа (response):

|  |  |
| --- | --- |
| HTTP/1.1 200 OK | VERSION STATUS\_CODE REASON |
| Content-Type: application/json Content-Length: 85  Server: nginx/1.18.0  Date: Tue, 02 Sep 2025 12:00 GMT Set-Cookie: sessionId=xyz789 | Header  (Header parameters)  keys: value |
|  | Пустая строка отделяющая header от body |
| {  "name": "John Doe",  "email": "john@example.com",  "age": 30,  "isActive": true  } | Body  Любой формат данных (может быть пустым) |

## Параметры HTTP запроса

По http запросы можно передать параметры несколькими способами

1) параметры пути (path parameters)

http:/www.google/{par1}/…

Самый простой способ передать параметры

В фаст апи можно передавать их как

@app.get("/todos/{id}/", response\_model=str)

def get\_two\_params(id: str):

    return f"id: {id}"

2) query

http:/www.google/?par1=1&par2=2

В принципе тоже самое тоже передаются в самом пути, но начинаются с ? и имеют наименование

В сваггере path и query также не будут отличаться, только названием внизу соответсвенно query или path

3) header parameters

Это параметр который не виден в пути и передаётся в header-e запроса:

Запрос с хедер параметрами:

curl -X 'GET' \

'http://localhost:8000/todos/' \

-H 'accept: application/json' \

-H 'token: qwe'

При этом -H 'accept: application/json' - формат файла в котором сервер должен прислать body ответа.

А -H 'token: qwe' - header параметр нашего запроса внутри файла который нам прислал сервер.

4) body параметры

Работает также как и с хеадер параметрами

## Подробнее про части http запроса

[Заголовки HTTP — Википедия](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BA%D0%B8_HTTP)

### HEADER

Header параметры в запросах - это метаданные, которые имеют свою классификацию на основе содержания хедера и четко определенную структуру, которая определена интернет стандартами ( напрмиер RFC - Internet Standard).

Вид хедера (string со структурой «название: значение»):

-H “smth: smth”

* При использовании друой структуры ( например -H "{'smth': 'smth'}" ) curl воспримет всю строку {'smth': 'smth'} как **имя заголовка**, а значение останется пустым.
* Название должно состоять минимум из одного печатного символа (ASCII-коды от 33 до 126). Регистр символов в названиях не имеет значения. Заголовки с неизвестными именами должны игнорироваться. После названия сразу должен следовать символ двоеточия.
* Значение может содержать любые символы ASCII кроме перевода строки (код 10) и возврата каретки (код 13). Пробельные символы в начале и конце значения обрезаются. Последовательность нескольких пробельных символов внутри значения может восприниматься как один пробел. Регистр символов также не имеет значения (если иное не предусмотрено форматом поля).

Инфо:

Такой хэдер является general header-ом, такие заголовки могут иметь любое содержание в рамках указанной структуры. Но существуют и другие классы хэдеров (вики).

**Стандарты формата символов хэдера и обработка сервером запросов с хэдером**

*Сериализация и отправка реквеста*

Для корректной сериализации запроса в *requests* вся строчка хэдера запроса должна состоять из симвлов ISO 8859-1 (Latin-1) – иначе будет ошибка latin-1. ISO 8859-1 включают в себя не только латинские буквы, но и управляющие символы и многие другие символы. Список всех ISO 8859-1 символов: <https://cs.stanford.edu/people/miles/iso8859.html#US-ASCII>

|  |
| --- |
| NOTE:  Даже запрос с управляющими и многими другими символами в хэдере может быть корректно сериализован и отправлен на сервер. Поэтому стоит хэндлить такие кейсы на сервере. |

*Стандарты обработки хэдеров*

**Однако,** помимо сериализации и отправки реквестов по адресу действует стандарт обработки хэдеров RFC 7230.

📜 Стандарт: [RFC 7230](<https://datatracker.ietf.org/doc/html/rfc7230>)

Это главный стандарт HTTP/1.1, и он говорит следующее:

Header field values MUST consist of printable characters: visible ASCII (0x20 – 0x7E), plus optional obs-text (bytes 0x80–0xFF), но не 0x00–0x1F (управляющие символы).

См. раздел [3.2 Field Parsing](<https://datatracker.ietf.org/doc/html/rfc7230#section-3.2>):

field-content = field-vchar [ 1\*( SP / HTAB ) field-vchar ]

field-vchar = VCHAR / obs-text

VCHAR = %x21-7E ; visible (printing) characters

obs-text = %x80-FF ; used only for backward compatibility

Это значит:

Допустимые символы: ASCII видимые символы 0x20 (пробел) до 0x7E (~).

И необязательные байты 0x80 – 0xFF (это ISO-8859-1).

Но никакие управляющие символы 0x00 – 0x1F, включая \x1b, не допускаются.

*Хенделинг севером невалидных символов заголовка запроса*

Я:

На сервер могут приходить недопустимые значения и имена хэдеров и я должен это хендлить (fast api автоматически возвращает 404 на такие запросы). Остаётся только задокументировать это в сваггере.

Deepseek:

FastAPI (точнее, его зависимость starlette) автоматически проверяет заголовки на соответствие стандартам HTTP:

* Имена заголовков должны быть ASCII-строками без управляющих символов (\x00-\x1F, \x7F).
* Значения заголовков не должны содержать запрещённых символов (например, \x1b).

Если клиент отправляет некорректные заголовки (например, с \x1b или не-ASCII без кодирования), FastAPI отклоняет запрос с 400, так как это потенциально опасный или нестандартный запрос.

Я:

Может в документации старлет сказано о том как конкртно она обрабатывает заголовки чтобы выбросить ошибку 400 <https://www.starlette.io/>

**Итого**

Назначение

Метаданные запроса или ответа. Они говорят что передаётся и как это интерпретировать.

Обработка протоколом

Жёстко регламентирован стандартами (RFC).

HTTP-протокол обязательно обрабатывает заголовки по установленным правилам.

Парсится как текст (ASCII), с чёткой структурой имя: значение.

Некоторые заголовки (например, Content-Length, Content-Type, Host) имеют особое значение, и нарушение их формата приведёт к ошибке.

### BODY

Body (-d for data) - это байты данных, передаваемые в сообщении HTTP-транзакции сразу после заголовков.

Формат в котором следует расшифровывать битовые данные определяется хэдером Content-Type

И вообще говоря могут быть любой структуры, главное указать формат битовых данных.

*Итого*

Назначение

Собственно содержимое (данные), передаваемое между клиентом и сервером (например, текст, JSON, файл, изображение и т.д.).

Обработка протоколом

Протокол не определяет чёткий формат тела — он просто передаёт "сырые байты".

Интерпретация зависит от заголовков (чаще всего от Content-Type).

Можно отправлять произвольные данные: текст, JSON, бинарные файлы, изображения и т.д.

**Детали**

## Форматы данных

### Биты -> файлы

Запрос пользователя и ответ всегда сериализуется в битовый формат, чтобы потом передать его по протоколу http. Поэтому очень важно согласовать тот как зашифровывать и расшифровывать такие бинарные файлы. Это обычно делают через описания в header (метаданные)

В зависимости от метода запроса хедеры будут разные, тк некоторые запросы не содержать body и сервер их не обрабатывает соответсвенно.

GET

-H “Accept: application/JSON” – формат файла, который сервер вернёт нам боди ответа.

POST etc

-H “Accept: application /JSON”

-H “Content-Type: application /JSON” – формат файла, который мы отправили в своем боди запроса

В фаст апи можно указать какой будет формат файла ответа сервера через response\_class, ДАЖЕ ЕСЛИ пользователь в своем запросе в хедере указал -H “Accept: application /JSON”:

@app.get("/todos/", response\_class=PlainTextResponse)

def get\_two\_params(par1: str):

    return par1

### Парсинг контента файлов

Помимо правильной расшифровки битного файла также важно записать контент этого файла в определенной структуре. Важно согласовать каким образом информация внутри файла будет структурирована, чтобы сервер распарсил боди запроса и я распарсил его боди ответа. Чаще всего это согласование происходит через сваггер/OpenAPI схему веб сервиса, которая показывает то какой эндпоинт в какой структуре возвращает данные, а другие пользователи на основе этой схемы могут обрабатывать ответы сервера.

Типы параметров в ответе сервера:

Чтобы формализовать структуру ответа сервера можно эксплицитно указать в коде, что сервер должен возращать результат в конкретном формате и сваггер автоматически опубликует это в схему. В фастапи это делается с помощью response\_model:

curl GET http:/example.com/todos/?par1=qwe

-H “Accept: PlainTextResponse”

@app.get("/todos/", response\_model=str)

def get\_two\_params(par1: str):

        return par1

При этом return функции обслуживающий эндпоинт должен быть согласован с указанным параметром Response\_model.

Типы параметров в запросе пользователя:

Чтобы формализовать струкутуру боди запроса пользователя указывают типы данных параметров запроса в аргументе функции, реализующий эндпоинт:

curl GET http:/example.com/todos/?par1=qwe/{item\_id}/

-H “Accept: PlainTextResponse”

-H “smth: smth”

-H “Content-Type: application /JSON”

-d “{‘smth2’: ‘smth2’}”

@app.get("/todos/{item\_id}", response\_model=str)

def get\_two\_params(par1: str, item\_id: int, head: str = Header(), body: MyModel = Body() ):

        return par1

Таким образом, фаст апи на основе этих данных может сформировать openAPI схему и сваггер, которые дадут полную информацию о том в каком формате пользователь должен присылать данные и в каком формате сервер отправить ответ. Это позволит надежно взаимодействовать с сервисом надстраивать другие сервисиы поверх нашего.

## Методы HTTP

1. Любой запрос содержить путь, хедер и тело. Поэтому даже GET эндпоинт может реализовать обработку тела запроса, но так не принято.

Таким образом, методы во многом несут семантическую функцию, хотя и GET и HEAD методы могут

**RFC 7231 — HTTP/1.1: Семантика и содержимое**  
Этот документ описывает, как серверы и клиенты должны интерпретировать различные HTTP-методы. Он подчёркивает, что HTTP — это **статeless** протокол прикладного уровня, и методы определяют лишь **намерения** клиента, но не влияют на то, как данные передаются по сети.   
**MDN Web Docs — HTTP Request Methods**  
В документации Mozilla объясняется, что методы HTTP, такие как GET, POST, PUT и DELETE, определяют действия, которые клиент хочет выполнить с ресурсом. Однако они не влияют на то, как данные передаются по сети.

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Основы организации контента сайта

### Html

#### Код для размещения элементов в контейнере

Порядок элементов / структура и иерархические отношения между элементами в целом детерминируются HTML кодом в jsx или tsx файле

Пример

<main className="container">

<h1 className="left-align">Проверка подключения к Oracle</h1>

<button onClick={testConnection} disabled={loading}>

{loading ? "Проверка..." : "Проверить подключение"}

</button>

{/\* Отображение результата \*/}

{message && <p>{message}</p>}

</main>

##### строка

По условию все элменты без каких либо контейнеров записываются в строчку

<main>

text1 text2

text1

text2

{text1 text2}

</main>

Всё это будет в одну строчку?

##### несколько строк

<main className="container">

<h1 className="left-align">Проверка подключения к Oracle</h1>

<button onClick={testConnection} disabled={loading}>

{loading ? "Проверка..." : "Проверить подключение"}

</button>

{/\* Отображение результата \*/}

{message && <p>{message}</p>}

</main>

Так:

элементом сверху будет текст

потом ниже кнопка

потом ниже сообщение

#### Структуры элементов

Можно сгруппировать элементы в один контейнер и задать для всех внутренних элементов один стиль и таким образом управлять визуалом программы

##### Контейнеры

Элементы в вебе обычно группируются в контейнеры

Основные типы контейнеров:

Семантические элементы для структурирования контента

<header>: Используется для создания заголовка страницы или раздела.

<footer>: Для создания футера страницы или раздела.

<main>: Определяет основное содержимое страницы.

<section>: Представляет собой独立的 раздел или группу контента.

<article>: Используется для контента, который может существовать независимо от остальной страницы (например, статьи, сообщения в блоге).

<aside>: Для контента, который补充основного содержимого (например, боковая панель с дополнительной информацией).

<nav>: Определяет навигационные ссылки.

<figure>: Используется для вставки изображений, диаграмм или других иллюстраций, которые могут иметь подпись <figcaption>.

Другие элементы

<blockquote>: Для цитат или выдержек из других источников.

<mark>: Выделяет текст, который должен быть выделен или помечен.

<code>: Используется для отображения блока кода или текста, который отличается от обычного текста

###### a) Базовые контейнеры:

<div> - базовый контейнер

<div> — универсальный контейнер без семантического значения. Используется для группировки элементов и применения стилей.

<div>

<p>Это параграф внутри div.</p>

<button>Кнопка</button>

</div>

<span> — inline-контейнер для текста или мелких элементов.

<p>Это <span style="color: red;">важный</span> текст.</p>

###### b) Семантические контейнеры:

Тоже самое что div, но имеют смысловое разделение, чтобы браузер мог понять, например <body> что это основная часть страницы и соответственно самая важная.

Семантические элементы для структурирования контента

<header>: Используется для создания заголовка страницы или раздела.

<footer>: Для создания футера страницы или раздела.

<main>: Определяет основное содержимое страницы.

<section>: Представляет собой独立的 раздел или группу контента.

<article>: Используется для контента, который может существовать независимо от остальной страницы (например, статьи, сообщения в блоге).

<aside>: Для контента, который补充основного содержимого (например, боковая панель с дополнительной информацией).

<nav>: Определяет навигационные ссылки.

<figure>: Используется для вставки изображений, диаграмм или других иллюстраций, которые могут иметь подпись <figcaption>.

<header> — шапка страницы или раздела.

<header>

<h1>Заголовок страницы</h1>

</header>

<main> — основное содержимое страницы.

<main>

<p>Основной контент страницы.</p>

</main>

<section> — отдельный раздел страницы.

<section>

<h2>Раздел 1</h2>

<p>Контент раздела.</p>

</section>

<article> — независимый контент (например, статья).

<article>

<h2>Заголовок статьи</h2>

<p>Текст статьи.</p>

</article>

<footer> — подвал страницы или раздела.

<footer>

<p>Подвал страницы.</p>

</footer>

###### c) Списки:

<ul>, <ol>, <li> — контейнеры для списков.

<ul>

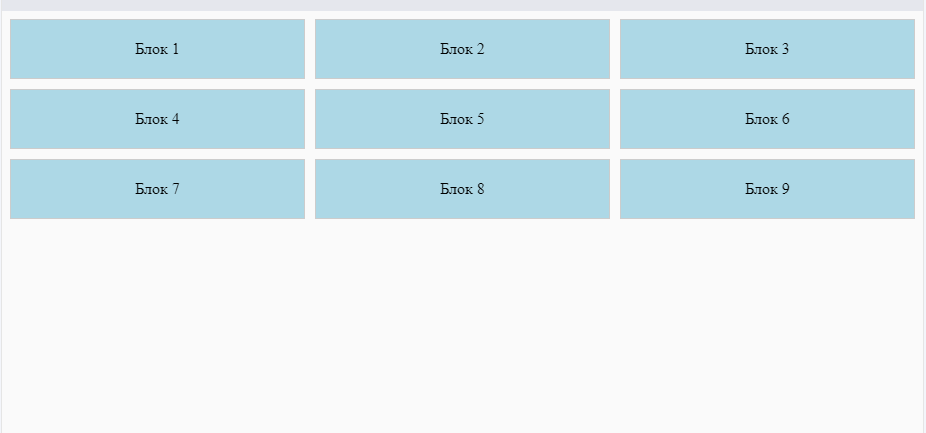
<li>Элемент 1</li>

<li>Элемент 2</li>

</ul>

##### Сетка блоков

По мимо вертикальных разбиений на части можно также разбить пространство на блоки, и в каждом блоке реализовать что то.



###### **CSS Grid Layout**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Grid Layout</title>

<style>

.grid-container {

display: grid;

grid-template-columns: repeat(3, 1fr); /\* 3 колонки, равной ширины \*/

grid-template-rows: auto auto auto; /\* 3 строки \*/

gap: 10px; /\* Расстояние между блоками \*/

}

.grid-item {

background-color: lightblue;

padding: 20px;

text-align: center;

border: 1px solid #ccc;

}

</style>

</head>

<body>

<div class="grid-container">

<div class="grid-item">Блок 1</div>

<div class="grid-item">Блок 2</div>

<div class="grid-item">Блок 3</div>

<div class="grid-item">Блок 4</div>

<div class="grid-item">Блок 5</div>

<div class="grid-item">Блок 6</div>

<div class="grid-item">Блок 7</div>

<div class="grid-item">Блок 8</div>

<div class="grid-item">Блок 9</div>

</div>

</body>

</html>

###### **Flexbox**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Flexbox Layout</title>

<style>

.flex-container {

display: flex;

flex-wrap: wrap; /\* Разрешаем перенос элементов на новую строку \*/

gap: 10px; /\* Расстояние между блоками \*/

}

.flex-item {

width: calc(33.333% - 10px); /\* 3 колонки с отступами \*/

background-color: lightgreen;

padding: 20px;

text-align: center;

border: 1px solid #ccc;

}

</style>

</head>

<body>

<div class="flex-container">

<div class="flex-item">Блок 1</div>

<div class="flex-item">Блок 2</div>

<div class="flex-item">Блок 3</div>

<div class="flex-item">Блок 4</div>

<div class="flex-item">Блок 5</div>

<div class="flex-item">Блок 6</div>

</div>

</body>

</html>

###### HTML Table (таблицы)

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Table Layout</title>

<style>

table {

width: 100%;

border-collapse: collapse;

}

td {

padding: 20px;

text-align: center;

border: 1px solid #ccc;

}

</style>

</head>

<body>

<table>

<tr>

<td>Блок 1</td>

<td>Блок 2</td>

<td>Блок 3</td>

</tr>

<tr>

<td>Блок 4</td>

<td>Блок 5</td>

<td>Блок 6</td>

</tr>

</table>

</body>

</html>

###### Реализация блочных элементов в react фреймворке

Вместо <html> в реакте используется <main classname = … >

Так main заменяет функцию html

function App() {

const [message, setMessage] = useState<string | null>(null);

const [loading, setLoading] = useState(false);

return (

<main className= "grid-container">

<div className="grid-item">

<h1 className="left-align">Количество изделий</h1>

<button onClick={testConnection} disabled={loading}>

{loading ? "Соединение..." : "Получить"}

</button>

{message && <p>{message}</p>}

</div>

<div className="grid-item">

<h1 className="left-align">Количество символов</h1>

<button onClick={symcount} disabled={loading2}>

{loading2 ? "Соединение..." : "Получить"}

</button>

{message2 && <p>{message2}</p>}

</div>

</main>

);

### СSS

#### Задать стиль всем элементам контейнера

.div1 {

display: flex;

flex-direction: column;

justify-content: center; /\* Центрирование по вертикали \*/

padding-left: 20%; /\* Отступ слева в процентах \*/

height: 100%; /\* Занимает всю высоту родительского контейнера \*/

/\*gap: 10px; /\* Отступ между элементами \*/

}

#### grid container

#### Управление сетки

.grid-container {

grid-template-columns

}

В строке будет 2 колонки

Первая колонка займёт 1 часть доступного пространства

Вторая колонка займёт 2 части доступного пространства

grid-template-columns: 1fr 2fr;

Фиксированное количество колонок (как у вас сейчас):

grid-template-columns: 1fr 1fr 1fr; /\* 3 колонки \*/

Автоматическое создание колонок:

grid-template-columns: repeat(auto-fill, 200px);

/\* Создаст столько колонок по 200px, сколько поместится в контейнер \*/

Гибкое автоматическое создание:

grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));

/\* Колонки будут минимум 200px, но растянутся чтобы заполнить пространство \*/

Фиксированная ширина с автоматическим переносом:

grid-auto-flow: dense;

grid-template-columns: repeat(5, 100px); /\* Всегда 5 колонок по 100px \*/

### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Решения для организации визуала

#### Модульный конструктор

Идея заключается в том, чтобы сделать main – Грид контейнером на весь экран (всегда был на весь экран).

И внутрь мейн грида добавить грид айтем (ячейки), которые будут играть роль модулей, которые реализуют свой функционал (внутри может быть любая структура) и они не зависят друг от друга.

Можно легко удалить ненужный и добавить новый.

##### Html

Архитектура:

<main className= "grid-container">

<div className= "grid-item">

<div className="div1">

<div className="row1">

<button onClick={testConnection} disabled={loading}>

{loading ? "Соединение..." : "Получить"}

</button>

{message && message}

</div>

<div className="row2">

<button onClick={symcount} disabled={loading2}>

{loading2 ? "Соединение..." : "Получить"}

</button>

{message2 && message2}

</div>

</div>

</div>

</main>

##### Css

Управление пространством внутри модуля

.div1 {

display: flex;

flex-direction: column;

justify-content: center; /\* Центрирование по вертикали \*/

padding-left: 20%; /\* Отступ слева в процентах \*/

height: 100%; /\* Занимает всю высоту родительского контейнера \*/

/\*gap: 10px; /\* Отступ между элементами \*/

}

# Описание стэка веб разработки

<https://www.youtube.com/watch?v=Tef1e9FiSR0>

1) html

html - Определяет содержимое и структуру страницы (например, заголовки, абзацы, изображения, ссылки)

2) css

внешний вид и оформление (цвета фон итд) (Не влияет на структуру или содержимое)

Более высокоуровневые языки для CSS для эффективного кода

2.1) CSS Preprocessors:

SASS

LESS

Stylus

2.2) CSS Framework:

Bootstrap

**Tailwind**

Foundation

Skeleton

3) JavaScript

JavaScript – современные приложение

4) TypeScript

TypeScript – статическая типизация и более мощный JS, более надеждный код, более предпочитается чем JS

5) UI toolkit

UI toolkit – react, Vue, Angular

# Различные архитектуры веб проекта

REST: Простые CRUD-приложения, публичные API.

gRPC: Высоконагруженные сервисы (микросервисы, внутренняя коммуникация).

GraphQL: Клиенты с динамическими требованиями к данным (например, фронтенд React).

AsyncAPI: Событийные системы (нотификации, обработка данных в реальном времени).

Если нужна максимальная скорость — gRPC.

Если нужна гибкость — GraphQL.

Если нужна простота — REST.

Если работаете с событиями — AsyncAPI.