

# JS Level 3

Node.js



# Node.js

В этой лекции мы с вами поговорим о построении API (Application Programming Interface).



# API



# API

API (Application Programming Interface) – программный интерфейс взаимодействия двух приложений. Определяет как две программы будут взаимодействовать между собой.

В нашем случае – это будет HTTP API: сервер с клиентом будут взаимодействовать посредством передачи сообщений по HTTP-протоколу.



# Задача

Наша задача – написать сервер, который умеет:

1. Отдавать список постов
2. Сохранять новый пост
3. Редактировать (сохранять обновление) существующего поста
4. Удалять пост



# CRUD

Такие системы называют CRUD-системами или говорят, что они поддерживают CRUD-операции.

CRUD (Create, Read, Update, Delete) – это те базовые операции, которые мы и перечислили.



# Задача

Итак, наш сервер должен в каком-то виде получать информацию от клиента, чтобы затем на основании этой информации выполнять нужные действия. И в то же время, он должен отдавать клиенту информацию (например, список постов), если клиент запрашивает именно это.

Давайте вспомним, какие сообщения передаются по протоколу HTTP.



# Messages

## 5 Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

```
Request      = Request-Line           ; Section 5.1
               *(( general-header      ; Section 4.5
                 | request-header      ; Section 5.3
                 | entity-header ) CRLF) ; Section 7.1
               CRLF
               [ message-body ]       ; Section 4.3
```

уже знакомые вам CRLF →

## 6 Response

After receiving and interpreting a request message, a server responds with an HTTP response message.

```
Response     = Status-Line           ; Section 6.1
               *(( general-header      ; Section 4.5
                 | response-header     ; Section 6.2
                 | entity-header ) CRLF) ; Section 7.1
               CRLF
               [ message-body ]       ; Section 7.2
```

уже знакомые вам CRLF →





# Request

Соответственно, в запросе у нас есть три варианта (включая их комбинации):

1. Request Line
2. Request Header
3. Request Body

Давайте начнём с первого.



# Request Line

Когда клиент делает HTTP-запрос, то он делает его на определённый URL вот в таком формате:

<http://host/path?query#fragment>, например:

<https://salom.alif.tj/partners?ce=114&#/partners>

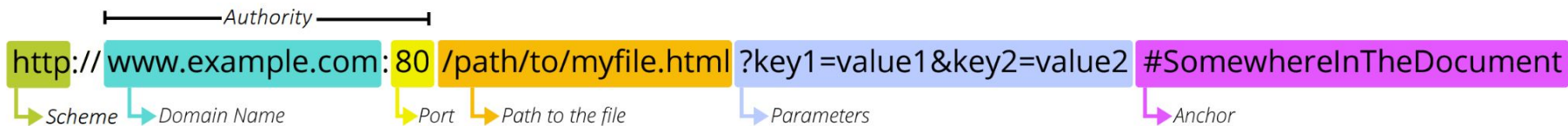
Где:

- **host** – это ip-адрес или имя домена (например, **salom.alif.tj**)
- **path** – это путь, например, **/partners**
- **query** – после **?** и до **#** в формате **key=value**, например, **?ce=114**
- **fragment** – после **#**, например **/partners**



# Request Line

Мы будем использовать термины **path**, **query** и **fragment**, хотя они могут иметь и другие названия:



# Request Line

Давайте напишем сервер, который будет с этим работать:

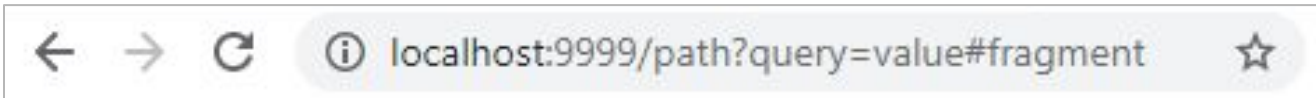
```
5  const server = http.createServer(function(request, response) {  
6  
7  });  
8  
9  server.listen(9999);
```

**Важно:** не забудьте про `'use strict'` и `require('node:http');` – для краткости, мы их не будем далее указывать



# Debug

Запустимся под debug'ом и отправим через браузер запрос вида:



# IncomingMessage

Первое, что нужно сделать, это перейти в документацию и [прочитать про класс](#)

`IncomingMessage*`: 

Среди свойств мы можем найти `url`: 

- `Class: http.IncomingMessage`
  - `Event: 'aborted'`
  - `Event: 'close'`
  - `message.aborted`
  - `message.complete`
  - `message.destroy([error])`
  - `message.headers`
  - `message.httpVersion`
  - `message.method`
  - `message.rawHeaders`
  - `message.rawTrailers`
  - `message.setTimeout(msecs[, callback])`
  - `message.socket`
  - `message.statusCode`
  - `message.statusMessage`
  - `message.trailers`
  - `message.url`

Примечание\*: всегда сначала читайте документацию, а не просто наугад пишете код.



# IncomingMessage

Описание самого класса:

**Class:** `http.IncomingMessage` #

► History

- Extends: `<stream.Readable>`

An `IncomingMessage` object is created by `http.Server` or `http.ClientRequest` and passed as the first argument to the `'request'` and `'response'` event respectively. It may be used to access response status, headers and data.



# IncomingMessage

Описание свойства **url**:

**message.url**

#

Added in: v0.1.90

- `<string>`

Only valid for request obtained from `http.Server`.

Request URL string. This contains only the URL that is present in the actual HTTP request. If the request is:

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

To parse the URL into its parts:

```
new URL(request.url, `http://${request.headers.host}`);
```





# Debug

Посмотрим на это свойство в дебаггере:

```
url: '/path?query=value'  
> __proto__: Readable  
> response: ServerResponse {_events: {.....  
> this: Server  
Return value: undefined
```

Всё так, как в документации – свойство типа **string**. Но работать с ним не удобно (не сами же мы будем разбирать этот запрос по частям), поэтому воспользуемся примером, который дан в документации (см. следующий слайд).



# URL

To parse the URL into its parts:

```
new URL(request.url, `http://${request.headers.host}`);
```

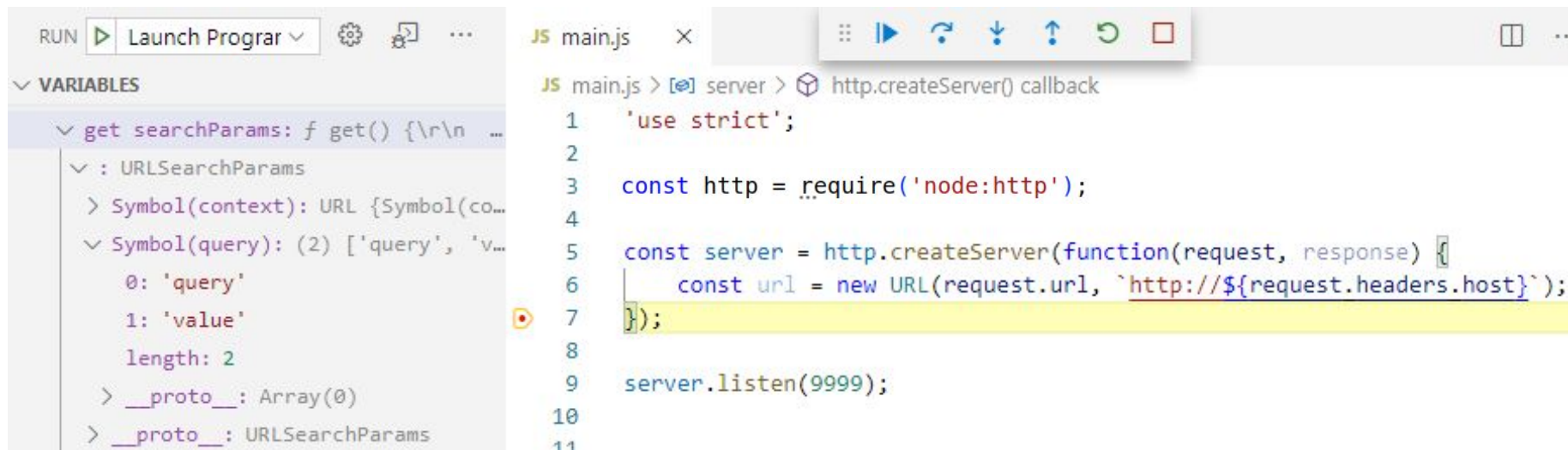
When `request.url` is `/status?name=ryan` and `request.headers.host` is `localhost:3000`:

```
$ node
> new URL(request.url, `http://${request.headers.host}`)
URL {
  href: 'http://localhost:3000/status?name=ryan',
  origin: 'http://localhost:3000',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'localhost:3000',
  hostname: 'localhost',
  port: '3000',
  pathname: '/status',
  search: '?name=ryan',
  searchParams: URLSearchParams { 'name' => 'ryan' },
  hash: ''
}
```



# URL

Внутри `url` откроем свойство `searchParams`:



The image shows a web browser's developer console on the left and a code editor on the right. The console displays the 'VARIABLES' section with a tree view of the `get searchParams` object. It is a `URLSearchParams` object with a `Symbol(query)` property containing an array of two elements: `'query'` and `'value'`. The `length` is 2. The `__proto__` is `Array(0)`. The `__proto__` is `URLSearchParams`. The code editor on the right shows a JavaScript file `main.js` with the following code:

```
JS main.js > [?] server > http.createServer() callback
1  'use strict';
2
3  const http = require('node:http');
4
5  const server = http.createServer(function(request, response) {
6    const url = new URL(request.url, `http://${request.headers.host}`);
7  });
8
9  server.listen(9999);
10
11
```

Теперь давайте разбираться.



# TEMPLATE LITERALS



# Template literals

Первое, это выражение ``http://${request.headers.host}`` (``` – backtick, там, где буква `ё`) – это строка, но не совсем обычная.

Символ ``` (в отличие от одинарных и двойных кавычек) позволяет нам:

1. Писать значение на нескольких строках
2. Использовать подстановку выражений

Что за подстановка выражений? Мы можем внутри строки прописать специальную конструкцию вида `${выражение}` и в строку "вклеится" результат вычисления этого выражения. В нашем случае – `localhost:9999`.



# Template literals

Подстановка значений – это не единственное, но самое часто используемое применение template literals. Поэтому мы активно будем его использовать.



# GET/SET



# get/set

Второе, это в дебаггере мы с вами видим приставки **get/set** рядом с именами свойств:

```
✓ url: URL {Symbol(context): URLCont...  
  > get hash: f get() {\r\n      cons...  
  > set hash: f set(hash) {\r\n      ...  
  > get host: f get() {\r\n      cons...  
  > set host: f set(host) {\r\n      ...
```

Давайте попробуем разобраться, что это значит. На время закомментируем наш сервер и напишем небольшое приложение.





# get/set

```
11 class Demo {  
12     get property() {  
13         console.log('get');  
14         return 'property';  
15     }  
16     set property(value) {  
17         console.log(`set ${value}`);  
18     }  
19 }  
20  
21 const demo = new Demo();  
22 const property = demo.property; // работает get  
23 demo.property = 'new value'; // работает set
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

C:\Program Files\nodejs\node.exe .\main.js

Debugger listening on ws://127.0.0.1:51526/88b321be-33d2-40be-9eca-db4cbc76a8ff

For help, see: <https://nodejs.org/en/docs/inspector>

Debugger attached.

get

set new value

Т.е. это возможность на операции чтения и записи свойства назначить функции.

При этом для пользователя класса это будет "не видно" – он будет работать с этими свойствами так же, как и до этого.



# get/set

Это достаточно мощная возможность, позволяющая вам выполнять какой-то код при чтении или записи свойств (при этом, поскольку это уже не просто запись свойства, а фактически, вызов функции, то идёт поиск по всей цепочке прототипов).

Например: можно запретить устанавливать неверные значения для свойства. Или сделать вид, что свойство существует (написав геттер), в то время как его реально может не быть.



# URL & URLSEARCHPARAMS



# URL & URLSearchParams

Осталось только разобраться, откуда берутся классы `URL` и `URLSearchParams`. Они берутся из модуля `url`. Но при этом мы не импортировали этот модуль, а просто использовали эти имена. Подумайте, как это возможно.

На самом деле, вы знаете ответ – эти имена есть в `global`. Открываем документацию на [Globals](#) (там перечислено всё, что в Node.js добавляется в `global`) и видим:

## URL

#

Added in: v10.0.0

The WHATWG `URL` class. See the `URL` section.

## URLSearchParams

#

Added in: v10.0.0

The WHATWG `URLSearchParams` class. See the `URLSearchParams` section.



# Query


Напоминаем, что часть **query** (которую описывает **URLSearchParams**) представляет собой набор пар ключ-значение:

**key=value1&key=value2**

Обратите внимание: в примере мы специально указали, что ключ может дублироваться. В этом случае, к одному ключу привязано несколько значений.



# URLSearchParams

Осталось только научиться с ними  
работать: 

Работа сведётся к нескольким  
методам:

1. Получить все параметры: **entries**
2. Получить по имени: **get**
3. Получить все значения  
параметра: **getAll**
4. Проверить, есть ли параметр: **has**
5. Получить все имена: **keys**
6. Получить все значения: **values**

- Class: `URLSearchParams`
  - Constructor: `new URLSearchParams()`
  - Constructor: `new URLSearchParams(string)`
  - Constructor: `new URLSearchParams(obj)`
  - Constructor: `new URLSearchParams(iterable)`
  - `urlSearchParams.append(name, value)`
  - `urlSearchParams.delete(name)`
  - `urlSearchParams.entries()`
  - `urlSearchParams.forEach(fn[, thisArg])`
  - `urlSearchParams.get(name)`
  - `urlSearchParams.getAll(name)`
  - `urlSearchParams.has(name)`
  - `urlSearchParams.keys()`
  - `urlSearchParams.set(name, value)`
  - `urlSearchParams.sort()`
  - `urlSearchParams.toString()`
  - `urlSearchParams.values()`
  - `urlSearchParams[Symbol.iterator]()`



# URLSearchParams

Например:

```
5  const server = http.createServer(function(request, response) {  
6    const url = new URL(request.url, `http://${request.headers.host}`);  
7    const searchParams = url.searchParams;  
8    response.end(searchParams.get('query'));  
9  });  
10  
11  server.listen(9999);
```

Обратите внимание: здесь **get** – это имя метода, а не геттер. Это первое. Второе: **get** возвращает первое значение ключа (т.е. если будет **query=value1&query=value2**, то вернёт **value1**) – это описано в документации.



# API





# API

Пришло время спроектировать наше API. Что значит спроектировать?

Это значит, решить: какие url'ы у нас будут в приложении, какие параметры мы будем требовать на вход, какими статус-кодами отвечать, в каком виде отвечать и т.д.



# API

Звучит немного сложно, не правда ли? Конечно же, можно взять умную книжку и начать читать, как делать "правильно". Но с таким подходом есть несколько проблем:

1. Книги пишутся так, чтобы было удобно показать конкретный способ решения проблемы
2. Не факт, что автор действительно сам проектировал API и был в этом успешен
3. И многие другие проблемы



# API

Поэтому мы предлагаем вам пойти немного другим путём. Вместо чтения книг (хотя их действительно нужно читать) мы будем смотреть, как это самое API реализуют в коммерчески успешных проектах.

Нужно отметить, что не всегда у них (у этих проектов) самое правильное, самое продуманное, самое удачное и удобное API. Но оно работает. И это ключевое: рабочий код всегда лучше красивого и правильного, но не рабочего.



# Vk API

Первым мы [рассмотрим Vk](#):

## Методы и объекты

API ВКонтакте — это интерфейс, который позволяет получать информацию из базы данных [vk.com](#) с помощью http-запросов к специальному серверу. Вам не нужно знать в подробностях, как устроена база, из каких таблиц и полей каких типов она состоит — достаточно того, что API-запрос об этом «знает». Синтаксис запросов и тип возвращаемых ими данных строго определены на стороне самого сервиса.

Например, для получения данных о пользователе с идентификатором *210700286* необходимо составить запрос такого вида:

```
https://api.vk.com/method/users.get?user_id=210700286&v=5.52
```

path

query



# Vk API

Рассмотрим отдельно все его составляющие.

- `https://` — протокол соединения.
- `api.vk.com/method` — адрес API-сервиса.
- `users.get` — название **метода** API ВКонтакте. Методы представляют собой условные команды, которые соответствуют той или иной операции с базой данных — получение информации, запись или удаление. Например, `users.get` — метод для получения информации о пользователе, `video.add` — метод для добавления видеозаписи в свой список, `likes.delete` — метод для удаления отметки «Мне нравится».

Все методы разделены на секции. Например, для работы с сообществами Вам нужны методы секции **groups**, для работы с фотографиями — **photos**, и так далее. Полный список методов по секциям доступен [на этой странице](#).

- `?user_id=210700286&v=5.52` — параметры запроса. После названия метода нужно передать его входные данные (если они есть) — как обычные GET-параметры в http-запросе. В нашем примере мы сообщаем серверу, что хотим получить данные о пользователе с `id=210700286` и формат этих данных должен соответствовать версии API 5.52 (о версиях мы еще поговорим позже). Входные параметры всегда перечислены на странице с описанием метода.



# Vk API

В ответ сервер вернет JSON-объект с запрошенными данными (или сообщение об ошибке, если что-то пошло не так). JSON — это формат записи данных в виде пар «имя свойства»: «значение». Если Вы раньше не встречались с этим форматом, мы рекомендуем познакомиться с ним, прежде чем продолжить чтение:

[JSON, Wikipedia](#) \*

Ответ на наш запрос выглядит так:

```
{"response":[{"id":210700286,"first_name":"Lindsey","last_name":"Stirling"}]}
```

Структура ответа каждого метода также строго задана, и при работе с API Вы заранее знаете, что в поле **id** придет число, а в поле **first\_name** — строка. Такие правила оговариваются на страницах с описанием метода и соответствующих объектов, которые он возвращает в ответе. Например, [users.get](#) — здесь описаны входные параметры метода и структура его ответа, а здесь — [user](#) подробно расписано каждое поле объекта из ответа.

Объект из ответа может быть не уникален для конкретного метода. Например, [объект пользователя](#) с набором полей, содержащих данные о его образовании, возрасте, интересах, может возвращаться в ответе от методов [users.get](#), [users.search](#), [groups.getMembers](#) и еще нескольких.



# Vk API

## Синтаксис запроса

Чтобы обратиться к методу API ВКонтакте, Вам необходимо выполнить **POST** или **GET** запрос такого вида:

```
https://api.vk.com/method/METHOD_NAME?PARAMETERS&access_token=ACCESS_TOKEN&v=V
```

Он состоит из нескольких частей:

- **METHOD\_NAME** (обязательно) — название метода API, к которому Вы хотите обратиться. Полный список методов доступен на [этой странице](#). **Обратите внимание:** имя метода чувствительно к регистру.
- **PARAMETERS** (опционально) — входные параметры соответствующего метода API, последовательность пар **name=value**, разделенных амперсандом. Список параметров указан на странице с описанием метода.
- **ACCESS\_TOKEN** (обязательно) — ключ доступа. Подробнее о получении токена Вы можете узнать в этом [руководстве](#).
- **V** (обязательно) — используемая версия API. Использование этого параметра применяет некоторые изменения в формате ответа различных методов. На текущий момент **актуальная версия API — 5.122**. Этот параметр следует передавать со всеми запросами.



# Vk API

## Wall

<code>wall.delete</code>	Удаляет запись со стены.
<code>wall.edit</code>	Редактирует запись на стене.
<code>wall.get</code>	Возвращает список записей со стены пользователя или сообщества.
<code>wall.getById</code>	Возвращает список записей со стен пользователей или сообществ по их идентификаторам.
<code>wall.getReposts</code>	Позволяет получать список репостов заданной записи.
<code>wall.pin</code>	Закрепляет запись на стене (запись будет отображаться выше остальных).
<code>wall.post</code>	Позволяет создать запись на стене, предложить запись на стене публичной страницы, опубликовать существующую отложенную запись.
<code>wall.repost</code>	Копирует объект на стену пользователя или сообщества.
<code>wall.search</code>	Позволяет искать записи на стене в соответствии с заданными критериями.





# Vk API

Wall > wall.get

Возвращает список записей со стены пользователя или сообщества.

## Параметры ^

<b>owner_id</b>	идентификатор пользователя или сообщества, со стены которого необходимо получить записи (по умолчанию — текущий пользователь).
<div>Обратите внимание, идентификатор сообщества в параметре <b>owner_id</b> необходимо указывать со знаком "-" — например, <b>owner_id=-1</b> соответствует идентификатору сообщества <a href="#">ВКонтакте API</a> (club1)</div>	
	целое число
<b>domain</b>	короткий адрес пользователя или сообщества. строка
<b>offset</b>	смещение, необходимое для выборки определенного подмножества записей. положительное число
<b>count</b>	количество записей, которое необходимо получить. Максимальное значение: 100. положительное число



# API

Теперь давайте обсуждать:

1. Нам нужно из **path** извлечь метод, например, **/method/wall.get**
2. Исходя из того, какой был метод – вызвать нужную функцию
3. Из query извлечь нужные параметры (параметры зависят от функции)
4. Нам нужно вернуть JSON (и разобраться, что это такое)
5. И ещё нужно обрабатывать ошибки (вдруг нам прислали не тот параметр)

И мы немного упростим, и будем считать, что у нас одна лента постов на всех.



# API

Начнём с самого простого: выберем **path** и **query**:

```
5  const server = http.createServer(function(request, response) {  
6    const url = new URL(request.url, `http://${request.headers.host}`);  
7    const pathname = url.pathname;  
8    const searchParams = url.searchParams;  
9  });  
10  
11  server.listen(9999);
```



# API

Теперь нужно исходя из `path` выбрать нужную функцию (ведь у нас будет, как минимум, 4).

Как это сделать? Можно, конечно, написать цепочку `if-else`, но это всегда плохая идея. Было бы здорово, если бы существовал какой-то класс вроде `URLSearchParams`, который бы умел искать по строке какие-то значения (в нашем случае – функции).

Это можно сделать с помощью обычных объектов (проверяя свойства на `undefined`), но давайте посмотрим на другие инструменты.



# MAP



# Map

В JS есть стандартный класс [Map](#), который позволяет хранить пары ключ-значение и обеспечивает функциональность, схожую с [URLSearchParams](#).

Главное отличие – ключи должны быть уникальны (т. е. не могут повторяться).

## Methods

```
Map.prototype.clear()
Map.prototype.delete()
Map.prototype.entries()
Map.prototype.forEach()
Map.prototype.get()
Map.prototype.has()
Map.prototype.keys()
Map.prototype.set()
Map.prototype.values()
Map.prototype[@@iterator]()
```



# Map

JS main.js > ...

```
1  'use strict';
2
3  const http = require('http');
4
5  const methods = new Map();
6  methods.set('/posts.get', function(request, response) {}); ← мы назвали posts, а не wall.
7  methods.set('/posts.getById', function(request, response) {});
8  methods.set('/posts.post', function(request, response) {});
9  methods.set('/posts.edit', function(request, response) {});
10 methods.set('/posts.delete', function(request, response) {});
11
12 const server = http.createServer(function(request, response) {
13   const url = new URL(request.url, `http://${request.headers.host}`);
14   const pathname = url.pathname;
15   const searchParams = url.searchParams;
16
17   const method = methods.get(pathname);
18   if (method === undefined) {
19     response.writeHead(404);
20     response.end();
21     return; ← early exit
22   }
23
24   method(request, response);
25 });
26
27 server.listen(9999);
```



# Map

JS main.js > ...

```
1  'use strict';
2
3  const http = require('http');
4
5  const port = 9999;
6  const statusNotFound = 404;
7
8  const methods = new Map();
9  methods.set('/posts.get', function(request, response) {});
10 methods.set('/posts.getById', function(request, response) {});
11 methods.set('/posts.post', function(request, response) {});
12 methods.set('/posts.edit', function(request, response) {});
13 methods.set('/posts.delete', function(request, response) {});
14
15 const server = http.createServer(function(request, response) {
16   const url = new URL(request.url, `http://${request.headers.host}`);
17   const pathname = url.pathname;
18
19   const method = methods.get(pathname);
20   if (method === undefined) {
21     response.writeHead(statusNotFound);
22     response.end();
23     return;
24   }
25
26   method(request, response);
27 });
28
29 server.listen(port);
```

Почистим код, убрав  
неиспользуемые  
переменные и вынеся  
числа в константы.





# POSTS



# Posts

Теперь нам нужно реализовать наши функции. Но чтобы их реализовать, нужно ответить на следующий вопрос: а где мы собираемся хранить посты?

Вариантов несколько:

1. В памяти (массив) – это значит, что перезапуская сервер, мы будем терять все накопленные посты
2. В файле
3. Используя базы данных (специализированное ПО)

Давайте начнём с первого варианта.



# Posts

Благодаря замыканиям, мы спокойно можем обращаться к `posts`:

```
5  const port = 9999;
6  const statusNotFound = 404;
7  const posts = [];
8
9  const methods = new Map();
10 methods.set('/posts.get', function(request, response) {
11   |   // TODO: work with posts
12 });
13 methods.set('/posts.getById', function(request, response) {});
14 methods.set('/posts.post', function(request, response) {});
15 methods.set('/posts.edit', function(request, response) {});
16 methods.set('/posts.delete', function(request, response) {});
17
18 > const server = http.createServer(function(request, response) { ...
30   });
31
32 server.listen(port);
```



# Posts

Теперь нужно обсудить, что же делать с ответом. В каком виде отдавать данные клиенту?



# JSON



# JSON

JSON (JavaScript Object Notation) – это специальный текстовый формат передачи данных, один из самых популярных на сегодняшний день.

В чём суть: программные системы могут быть написаны на разных языках, например, мы можем писать фронтенд на JS, а мобильные приложения пишутся на Kotlin или Swift. При этом всем этим системам нужно обмениваться данными. JSON – как раз-таки такой формат, который позволяет это делать.



# JSON

В общем виде JSON представляет собой документ, в котором могут быть представлены следующие типы данных:

- строки (string)
- числа (number)
- boolean
- null
- объекты
- массивы (выделяется отдельно, хотя в рамках JS массив - это тоже объект)

Нет ни **Undefined**, ни **BigInt**. Т.е. передаются не JS-объекты, а их представление в формате JSON. При этом можно передавать только данные (функции передавать нельзя).



# JSON

Сам формат – текстовый, т.е. мы можем его увидеть в виде текста. В стандартной библиотеке JS есть глобальный объект **JSON**, у которого есть два ключевых метода:

- **stringify**: из JS объекта делает JSON-документ
- **parse**: из JSON-документа делает JS-объект





# JSON

Чтобы клиент понял, что ему отдают JSON, а не просто текст, нужно выставить заголовок `application/json`.



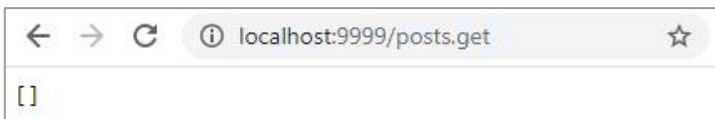
# JSON

```
5  const port = 9999;
6  const statusNotFound = 404;
7  const statusOk = 200;
8  const posts = [];
9
10 const methods = new Map();
11 methods.set('/posts.get', function(request, response) {
12     response.writeHead(statusOk, {'Content-Type': 'application/json'});
13     response.end(JSON.stringify(posts));
14 });
15 methods.set('/posts.getById', function(request, response) {});
16 methods.set('/posts.post', function(request, response) {});
17 methods.set('/posts.edit', function(request, response) {});
18 methods.set('/posts.delete', function(request, response) {});
19
20 > const server = http.createServer(function(request, response) { ...
32     });
33
34 server.listen(port);
```



# JSON

Смотрим в браузере:



Получили пустой массив, так, как и должны были.



# JSON

Чуть позже мы устраним дублирование кода, пока же давайте реализуем добавление.



**ДОБАВЛЕНИЕ**



# Добавление

Мы сами решаем, какие параметры должны быть у поста. Пусть для простоты это будет просто свойство `content`.

Но при этом сам `post` должен содержать следующие свойства:

- `id` – уникальный идентификатор
- `content` – содержимое поста
- `created` – дата создания

Сервер самостоятельно должен рассчитывать `id` и дату создания.



# id

**id** – это уникальный идентификатор, который позволяет найти объект среди всех других объектов такого же типа. Может быть числом, может быть строкой. Вы должны привыкнуть, что у большинства сущностей, которые вы собираетесь хранить, будет **id**.

Самый простой – это число. И мы можем при добавлении нового поста просто увеличивать это число на **1**. Для этого нам где-то нужно хранить следующее значение **id**. Так и назовём его **nextId**.



```

5  const port = 9999;
6  const statusOk = 200;
7  const statusBadRequest = 400; ← клиент прислал "плохой" запрос
8  const statusNotFound = 404;
9
10 let nextId = 1;
11 const posts = [];
12
13 const methods = new Map();
14 methods.set('/posts.get', function(request, response) {
15     response.writeHead(statusOk, {'Content-Type': 'application/json'});
16     response.end(JSON.stringify(posts));
17 });
18 methods.set('/posts.getById', function(request, response) {});
19 methods.set('/posts.post', function(request, response) {
20     const url = new URL(request.url, `http://${request.headers.host}`);
21     const searchParams = url.searchParams;
22
23     if (!searchParams.has('content')) {
24         response.writeHead(statusBadRequest);
25         response.end();
26         return;
27     }
28
29     const content = searchParams.get('content');
30
31     const post = {
32         id: nextId++,
33         content: content,
34         created: Date.now(),
35     };
36
37     posts.unshift(post); ← добавляем созданный пост в начало массива
38     response.writeHead(statusOk, {'Content-Type': 'application/json'});
39     response.end(JSON.stringify(post)); ← отправляем клиенту созданный пост
40 });
41 methods.set('/posts.edit', function(request, response) {});
42 methods.set('/posts.delete', function(request, response) {});

```





# nextId++

**++** – это оператор инкремента. Может писать перед переменной (тогда называется префиксным), либо после (тогда называется постфиксным).

Чаще всего используется постфиксный вариант.

Выражение **nextId++** работает следующим образом:

1. В качестве результата выражения используется текущее значение (**1**)
2. Значение **nextId** увеличивается на **1** (становится **2**)

Когда добавляется второй пост:

1. В качестве результата выражения используется текущее значение (**2**)
2. Значение **nextId** увеличивается на **1** (становится **3**)



# Date.now()

[Date](#) – ещё один глобальный объект, который позволяет работать с датой. При работе с датой и временем есть одна большая проблема – дата и время разные в зависимости от вашего часового пояса. Поэтому, чтобы упростить себе жизнь, разработчики договорились ввести специальный термин (чаще всего его называют **timestamp**, **epoch timestamp**, **unix timestamp**) – количество секунд, прошедших с 1 января 1970 года в рамках UTC (всемирного скоординированного времени). Оно одинаковое для всех, вне зависимости от часового пояса.

В JS объект **Date.now()** возвращает это количество, но только не секунд, а миллисекунд.



# JSON

Смотрим в браузере:



A screenshot of a web browser's address bar and content area. The address bar shows the URL `localhost:9999/posts.post?content=first` with navigation icons (back, forward, refresh) and a star icon. The content area displays a JSON object: `{"id":1,"content":"first","created":1597979237263}`.

Теперь пробуем получить список всех постов:



A screenshot of a web browser's address bar and content area. The address bar shows the URL `localhost:9999/posts.get` with navigation icons (back, forward, refresh) and a star icon. The content area displays a JSON array: `[{"id":1,"content":"first","created":1597979237263}]`.



# РЕФАКТОРИНГ



# Код

Наш код достаточно плох, потому что содержит много дублирующихся строк. Например, мы уже несколько раз писали одинаковый заголовок 200 и делали `JSON.stringify`. А создание URL'a мы вообще повторили два раза – в `createServer` и функции `/posts.post`.

Давайте улучшать. Обратите внимание: мы сначала добились того, чтобы наш код работал (оставшиеся 3 функции вам нужно будет сделать в рамках ДЗ), а потом только начали улучшать.



# Рефакторинг

Рефакторинг – это улучшение структуры нашего кода, без изменения его функциональности. Т.е. как бы вам не хотелось в процессе улучшения структуры сразу ещё добавить новых возможностей – не стоит этого делать.

Кроме того, прежде чем что-то улучшать, неплохо бы сохранить резервную копию. Для этого нам и нужен будет Git.



# Git



# Git

Для установки Git перейдите по адресу <https://git-scm.com/downloads> и выберите установочный файл для вашей операционной системы (далее – ОС). Обычно, ОС определяется автоматически и вам нужно лишь нажать на кнопку скачивания:





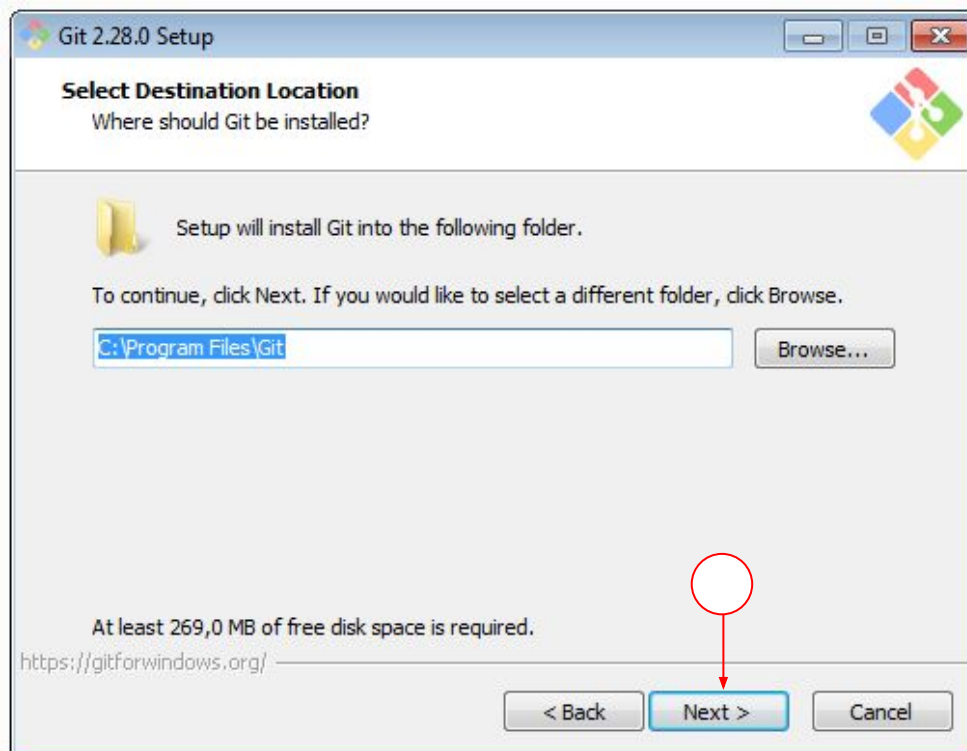
# Git

Дождитесь загрузки файла и откройте его, прочитайте и согласитесь с условиями лицензии (нажмите кнопку "Next"):



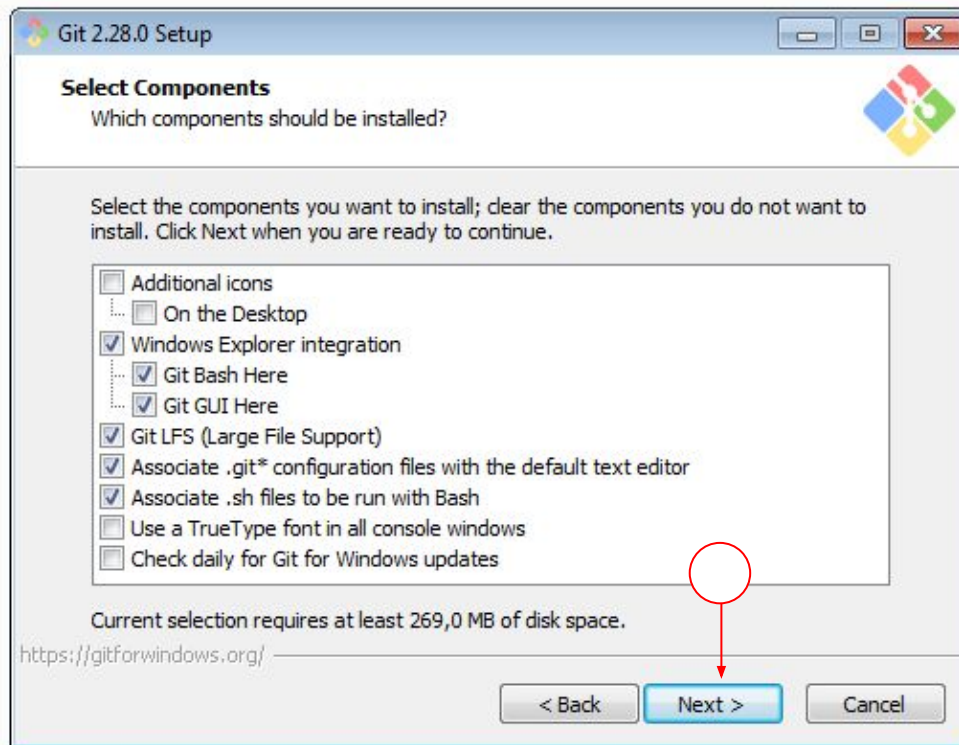
# Git

Подтвердите установку в указанный каталог:



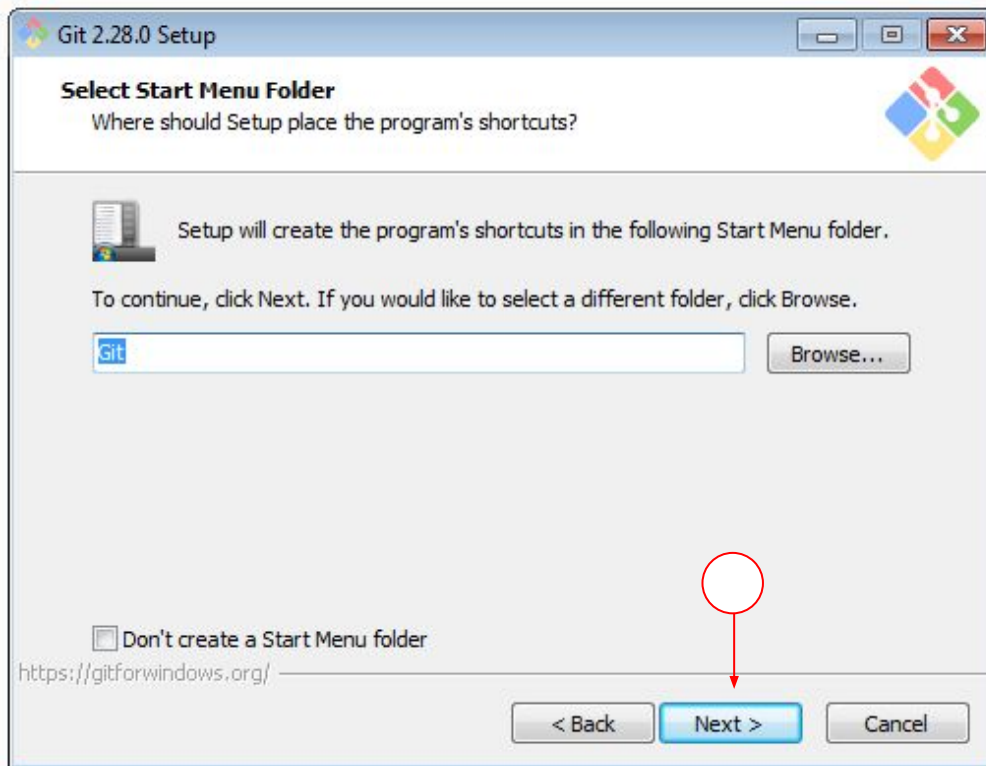
# Git

Оставьте выбранные значения по умолчанию и нажмите на кнопку "Next":



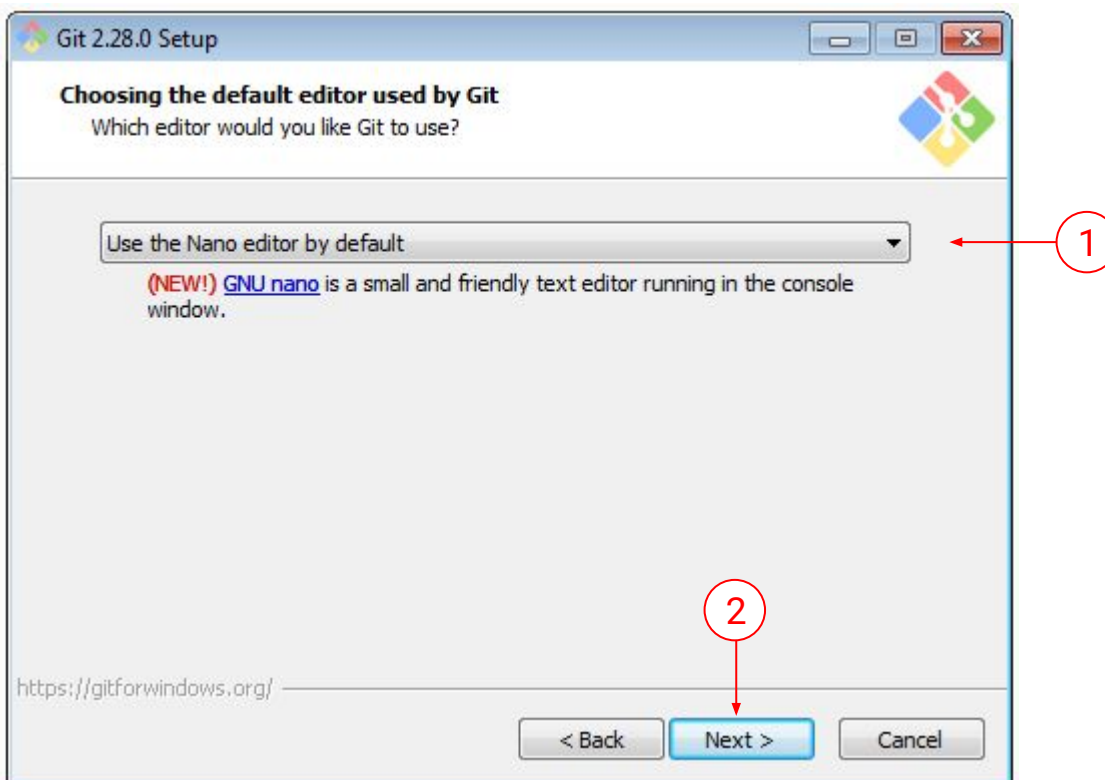
# Git

Оставьте выбранные значения по умолчанию и нажмите на кнопку "Next":



# Git

В следующем окне выберите Nano вместо Vim (вы можете оставить Vim, если захотите научиться им пользоваться) и нажмите "Next".



# Vim

Vim – это мощный текстовый редактор, позволяющий вам максимально эффективно работать с текстом (кодом в том числе). Он достаточно тяжёл для освоения, но если вы научитесь им пользоваться, то ваша скорость и эффективность работы с текстом и кодом вырастет в несколько раз.

Плагины, обеспечивающие функциональность Vim есть для всех популярных редакторов кода – от VS Code до онлайн-редакторов.



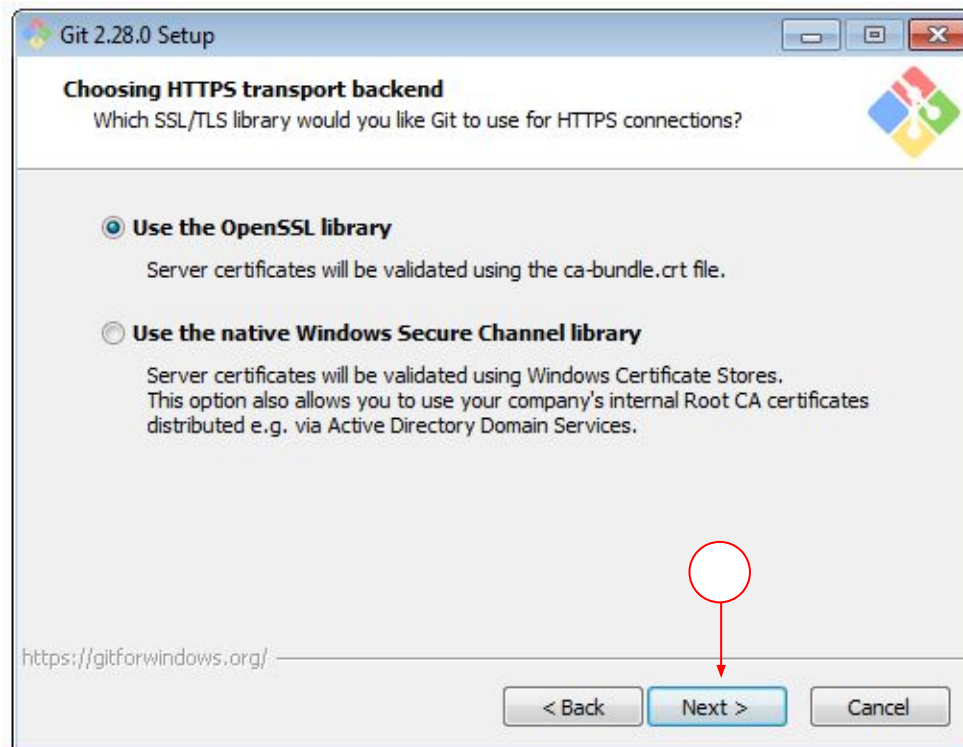
# Git

Оставьте выбранные значения по умолчанию и нажмите на кнопку "Next":



# Git

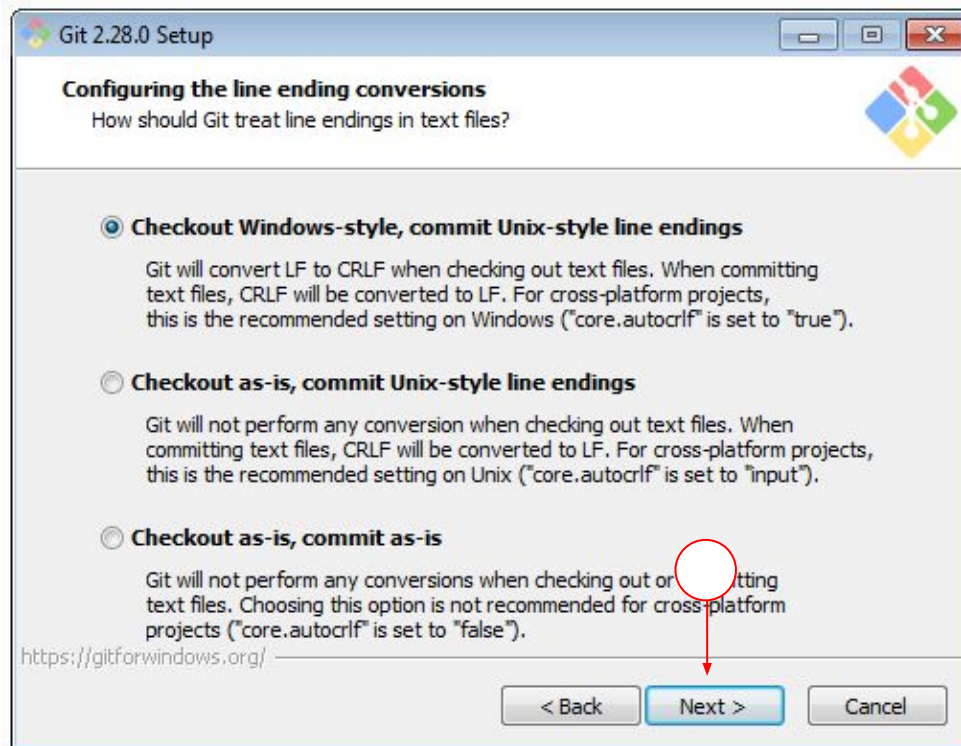
Оставьте выбранные значения по умолчанию и нажмите на кнопку "Next":





# Git

Оставьте выбранные значения по умолчанию и нажмите на кнопку "Next":



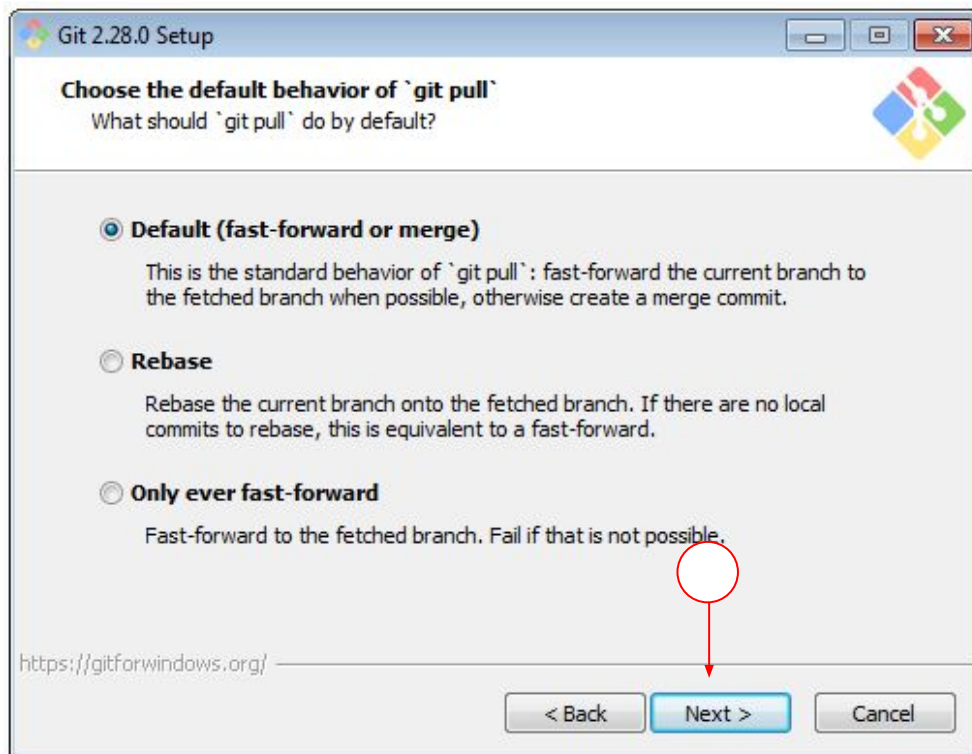
# Git

Оставьте выбранные значения по умолчанию и нажмите на кнопку "Next":



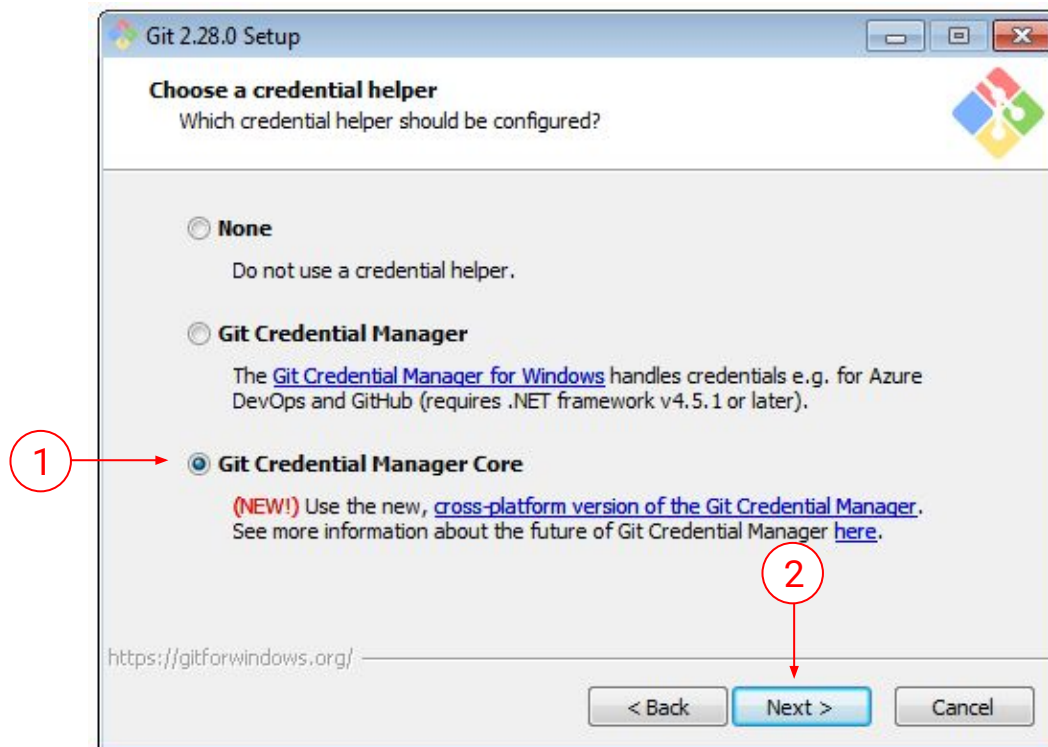
# Git

Оставьте выбранные значения по умолчанию и нажмите на кнопку "Next":



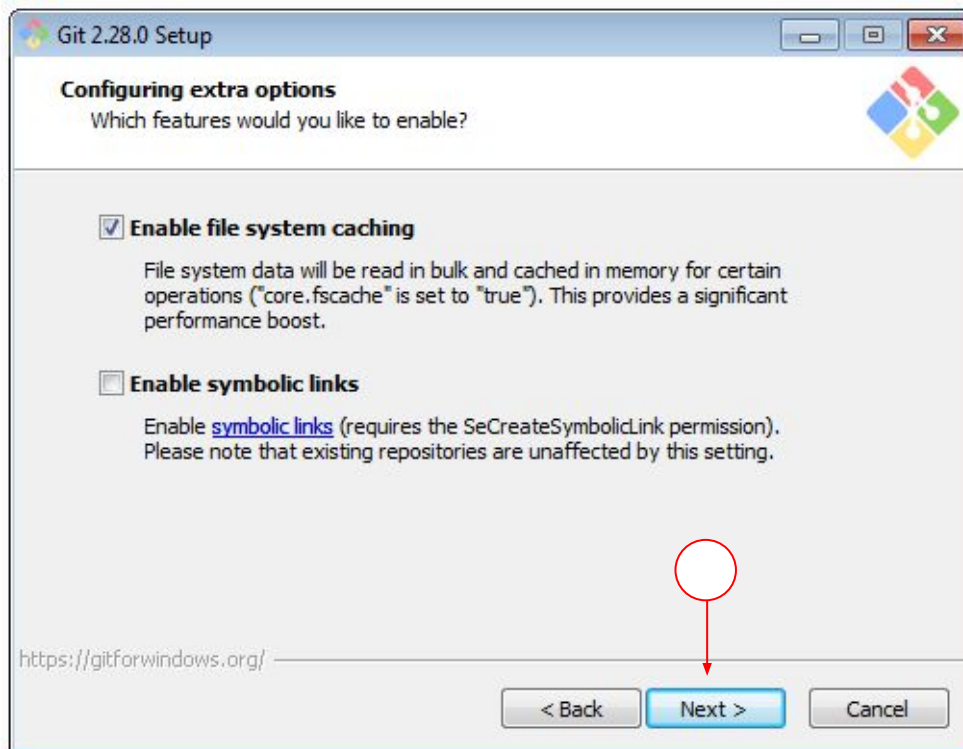
# Git

Выберите Git Credential Manager Core и нажмите на кнопку "Next".



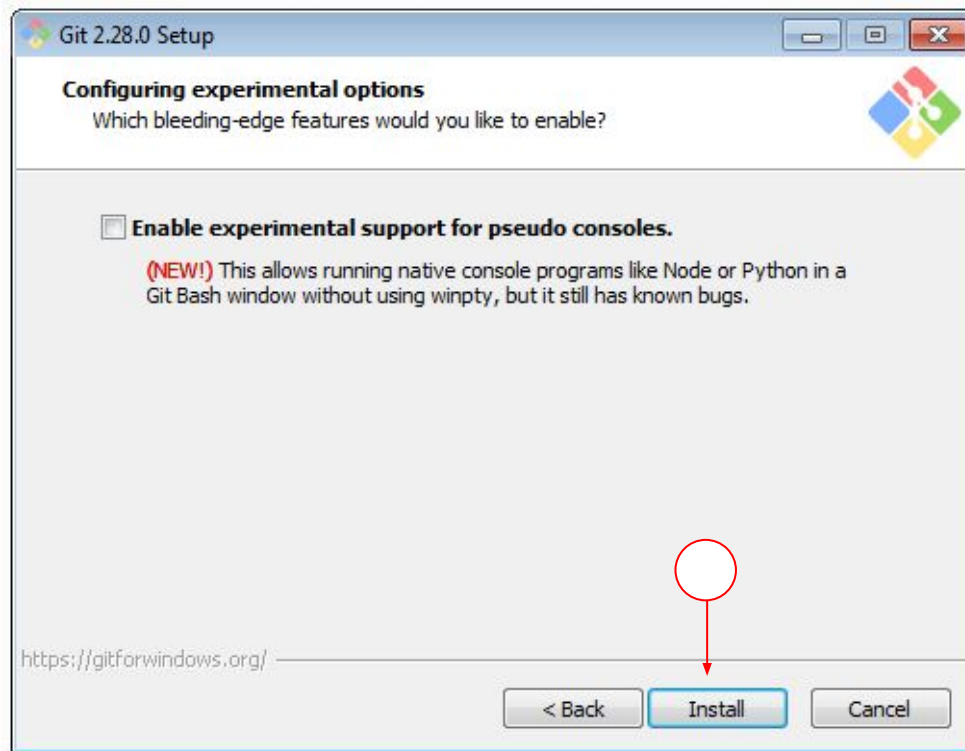
# Git

Оставьте выбранные значения по умолчанию и нажмите на кнопку "Next":



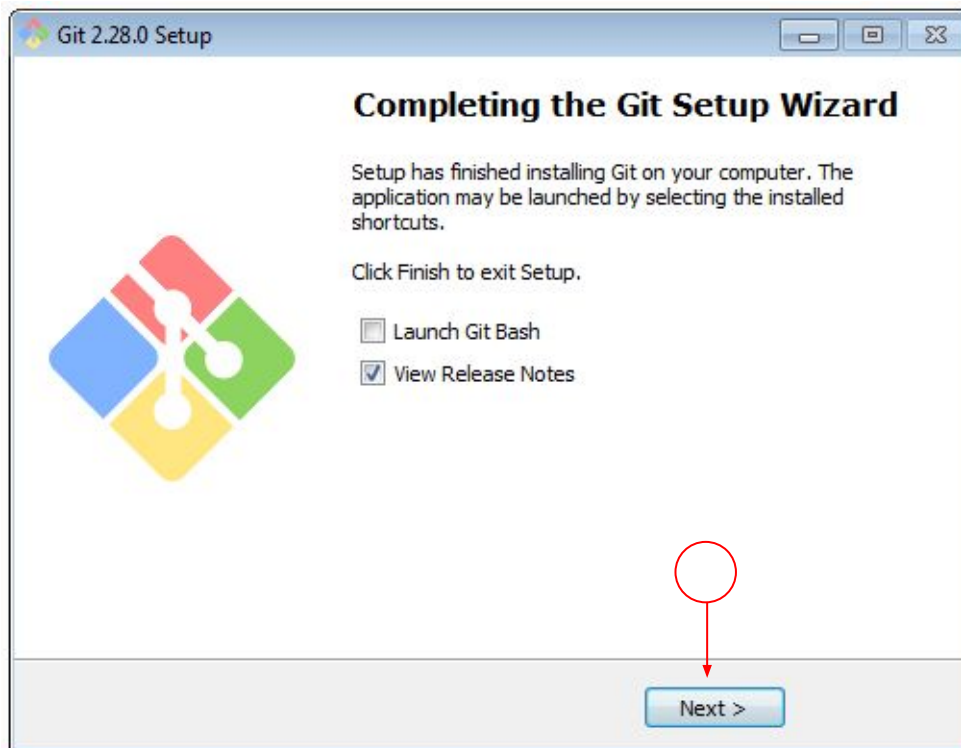
# Git

Оставьте выбранные значения по умолчанию и нажмите на кнопку "Install":



# Git

Дождитесь завершения установки и нажмите кнопку "Next":



# Git

В консоли VS Code (открывается по **Ctrl + `**, ` – там, где буква ё на клавиатуре)

введите команду **git --version**:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Microsoft Windows [Version 6.1.7601]
```

```
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.
```

```
C:\projects\intro>git --version
```

```
"git" не является внутренней или внешней
```

```
командой, исполняемой программой или пакетным файлом.
```

Если вы видите такое сообщение, значит вы либо неправильно установили Git, либо не перезапустили VS Code после установки Git.





# Git

Если вы видите такое сообщение (версия может быть новее), значит всё хорошо (и мы можем приступить к работе):

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\projects\intro>git --version
git version 2.28.0.windows.1
```



# Terminal

Обратите внимание, в оболочках CMD, PowerShell символ **>** означает приглашение к вводу команды (ввод команды завершается нажатием клавиши **Enter**):

```
C:\projects\intro>■
```

```
PS C:\projects\intro> ■
```

В оболочках Bash, Zsh и другие, используется символ **\$** (это настраивается):



# Terminal

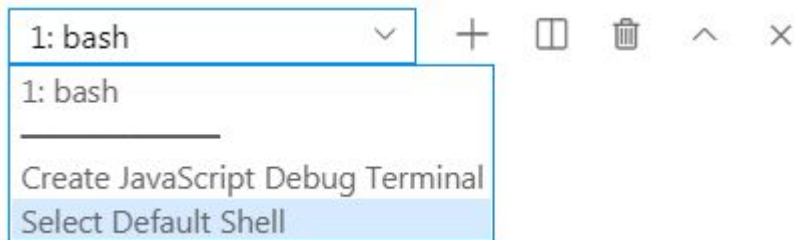
Терминал (консоль) – это специальный интерфейс, в котором вы в текстовом виде вводите команды, а операционная система выдаёт вам результат выполнения этих команд.

У нас есть консоль в браузере, а это – прямо в вашей операционной системе.



# Terminal

В VS Code вы всегда можете выбрать оболочку по душе, выбрав в панельке терминала:



Кнопка + позволяет создать новую вкладку с терминалом

Кнопка 🗑️ позволяет закрыть текущий терминал



# GitHub



# GitHub

GitHub – это централизованная платформа для разработчиков. Она предоставляет вам возможность хранить копии своих репозиториев онлайн.

Перейдите по адресу: <https://github.com/join> и заполните поля (см. следующий слайд).



# GitHub

Join GitHub

## Create your account

Username \*

coursar



Email address \*

coursar@email.com



Password \*

.....



Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. [Learn more.](#)

**Важно:** скриншот страницы может отличаться (например, будет чёрный экран на фоне планеты) – разработчики GitHub периодически меняют дизайн.



# GitHub

Кроме того, вам могут предложить решить пазл или воспользоваться другим механизмом, чтобы подтвердить, что вы не робот:

Verify your account

Pick the dice pair adding up to 5





# GitHub

После этого активируется кнопка Create Account:

Verify your account





Create account






# GitHub


На почту вам придёт вот такое письмо:

 Несортированные

 Соцсети

 Промоакции

<input type="checkbox"/>	 GitHub	[GitHub] Please verify your email address.	20:36
<input type="checkbox"/>	 Команда Google	Мария, завершите настройку нового ак...	20:22



Almost done,                      ! To complete your  
GitHub sign up, we just need to verify your  
email address:                      .

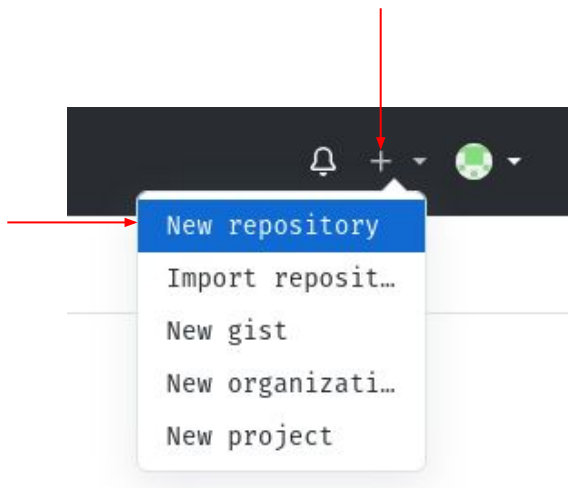
Verify email address

нужно кликнуть



# GitHub




Далее ищите в правом верхнем углу кнопку + и выбираете Repository:



# GitHub

Заполните поле Repository name как на экране и нажмите Create Repository:


Owner \* Repository name \*


  /   

Great repository names are short and memorable. Need inspiration? How about [ubiquitous-dollop?](#)

Description (optional)

---

☒  **Public**  
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**  
You choose who can see and commit to this repository.

---


**Initialize this repository with:**  
Skip this step if you're importing an existing repository.

☐ **Add a README file**  
This is where you can write a long description for your project. [Learn more.](#)

☐ **Add .gitignore**  
Choose which files not to track from a list of templates. [Learn more.](#)

☐ **Choose a license**  
A license tells others what they can and can't do with your code. [Learn more.](#)

---





# Основы работы



# VCS

Version Control System (VCS) – система контроля версий, предназначена для отслеживания изменений и коллективной работы.

Т.е. это отдельная программа, которая запоминает состояние ваших файлов (их содержимое) на определённый момент времени (как фото). И вы всегда можете вернуться, посмотреть, что было раньше, сравнить с тем, что есть сейчас.



# VCS

Например, такое часто бывает, когда вы делаете доклад и согласовываете его с преподавателем. У вас появляется куча версий одного и того же документа, но вы нумеруете их вручную:

	Доклад (версия 1) - правки	Дата изменения: 09.08.2020 13:31 Размер: 0 байт
	Доклад (версия 1)	Дата изменения: 09.08.2020 13:31 Размер: 0 байт
	Доклад (итоговый)	Дата изменения: 09.08.2020 13:31 Размер: 0 байт
	Доклад (финальная версия)	Дата изменения: 09.08.2020 13:31 Размер: 0 байт
	Доклад	Дата изменения: 09.08.2020 13:31 Размер: 0 байт

Конечно же, через какое-то время вы обязательно запутаетесь – какая версия была последней?



# VCS

А самое проблемное – даже если вы будете аккуратно хранить эти файлы, то ничто не мешает вам их случайно удалить, или заменить самую последнюю версию на одну из старых версий.

VCS как раз и призваны решить эту проблему – вы им даёте команду запомнить текущее содержимое вашего каталога (и они запоминают). Не нужно никаких ручных именований и прочего.





# VCS

Существуют разные системы контроля версий: Git, SVN (Subversion), Mercurial и другие.

Но стандартом де-факто сейчас является Git, поэтому изучать мы будем именно его.



# VCS

Чтобы детальнее разобраться со всем, давайте посмотрим, как работают VCS и какие они бывают.

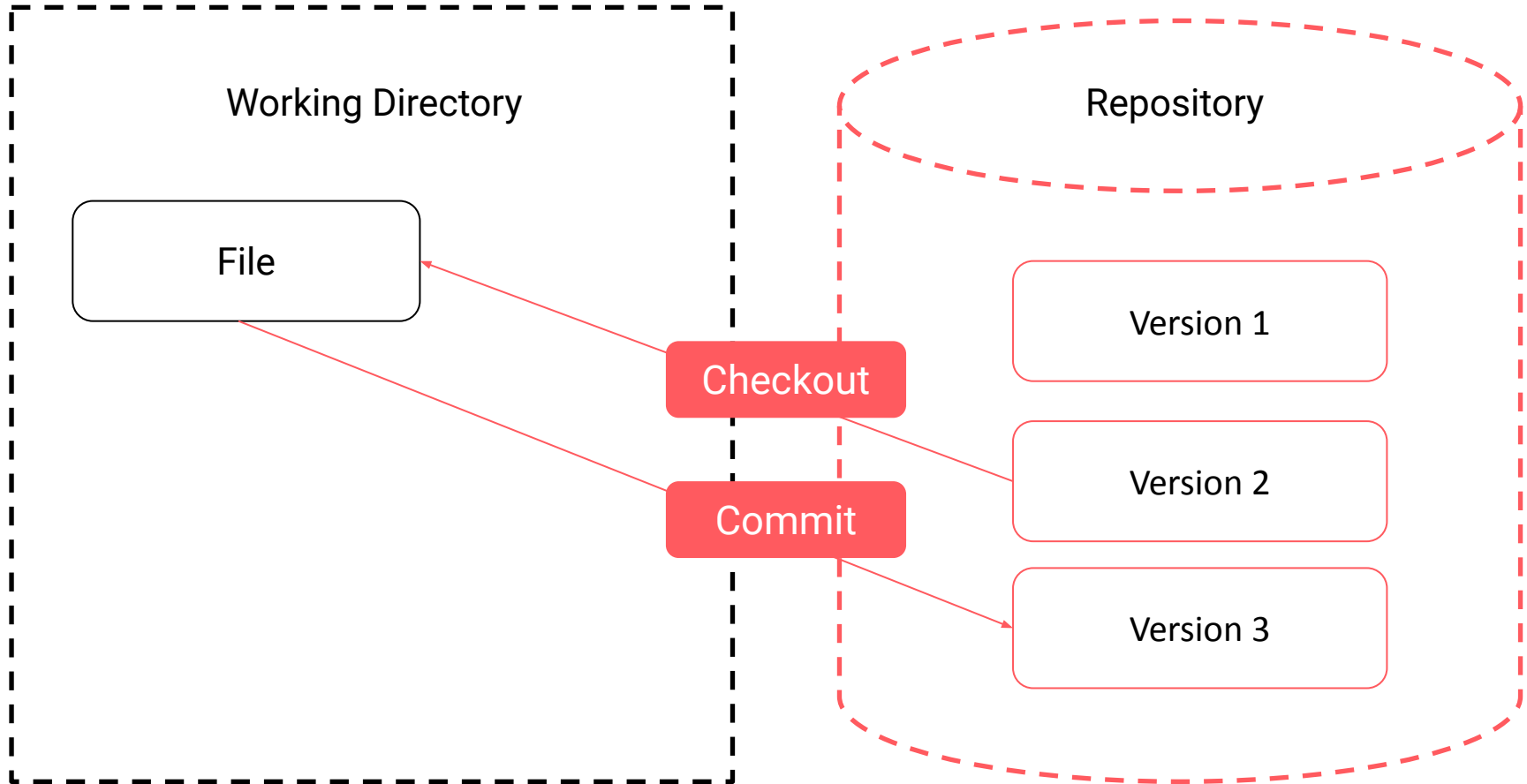
Для этого нам нужно ввести несколько понятий:

- рабочий каталог – это просто каталог, в котором хранятся ваши файлы (в нашем примере – `C:\projects\intro`)
- репозиторий – это некая "магическая" штука, которая умеет запоминать все файлы\* из вашего каталога на определённый момент времени

Примечание\*: можно заставить запоминать только некоторые.



# VCS



# Commit & Checkout

У репозитория есть две ключевых функции:

1. `commit` – запомнить состояние вашего рабочего каталога (это значит сохранить всё содержимое файлов, присвоить версию)
2. `checkout` – выдать вам содержимое рабочего каталога определённой версии (которая была "запомнена" с помощью `commit`)



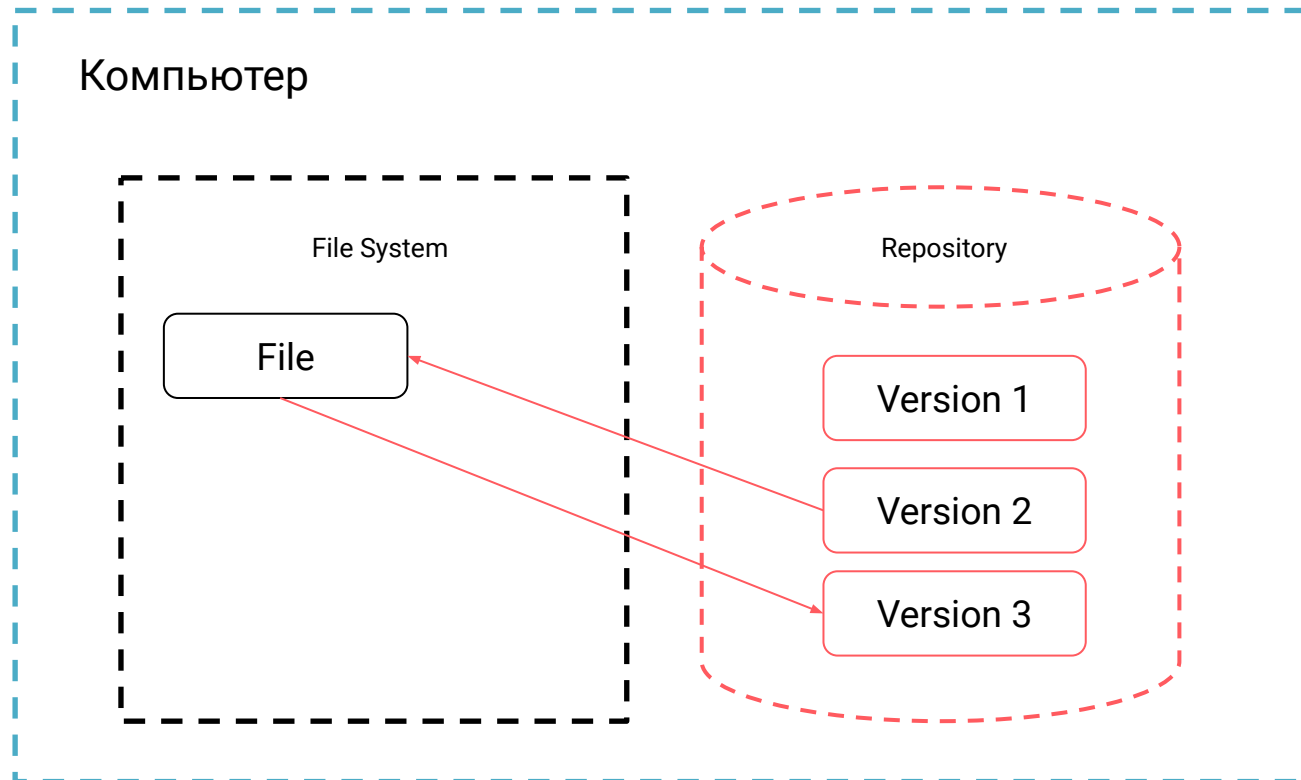
# VCS

VCS бывают трёх типов:

- локальные
- централизованные
- распределённые



# Local VCS



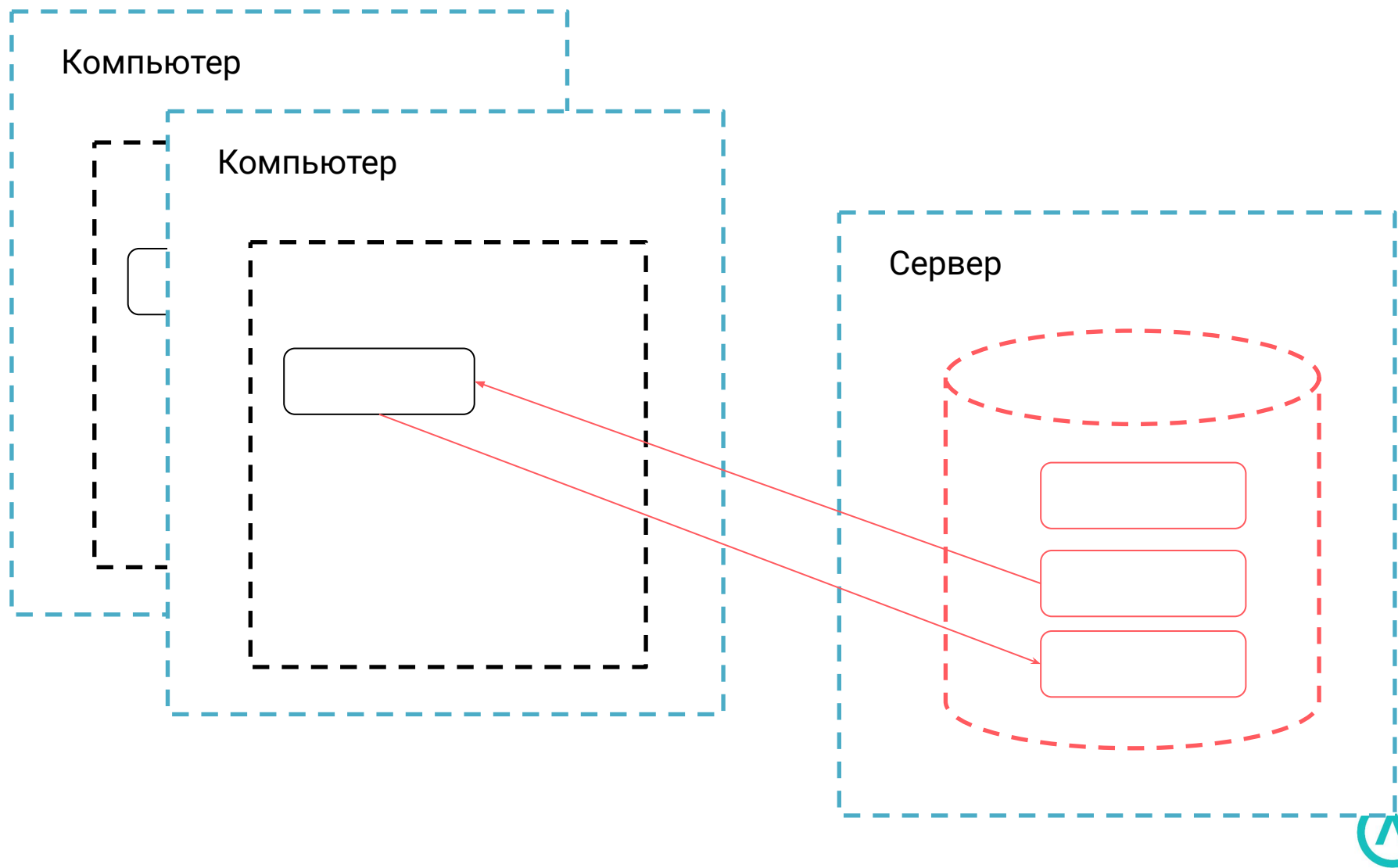
# Local VCS

В случае локальной VCS весь репозиторий хранится на том же компьютере, что и рабочий каталог. А это значит, что если что-то с вашим компьютером случится, то вся история работы с проектом (и сам проект) пропадут (типичный синдром студента – за день до сдачи сломался компьютер со всеми файлами курсовой/диплома).

Но с другой стороны, есть и плюсы – вам не нужен Интернет для работы (либо сетевое подключение к другому компьютеру).



# Centralized VCS





# Centralized VCS

В случае централизованной VCS весь репозиторий хранится на отдельном компьютере, который называется сервером. Этот сервер, по возможности, должен быть всегда включен и подключен к сети.

На рабочих компьютерах пользователей хранятся только файлы конкретной версии (с которыми они сейчас работают).



# Centralized VCS

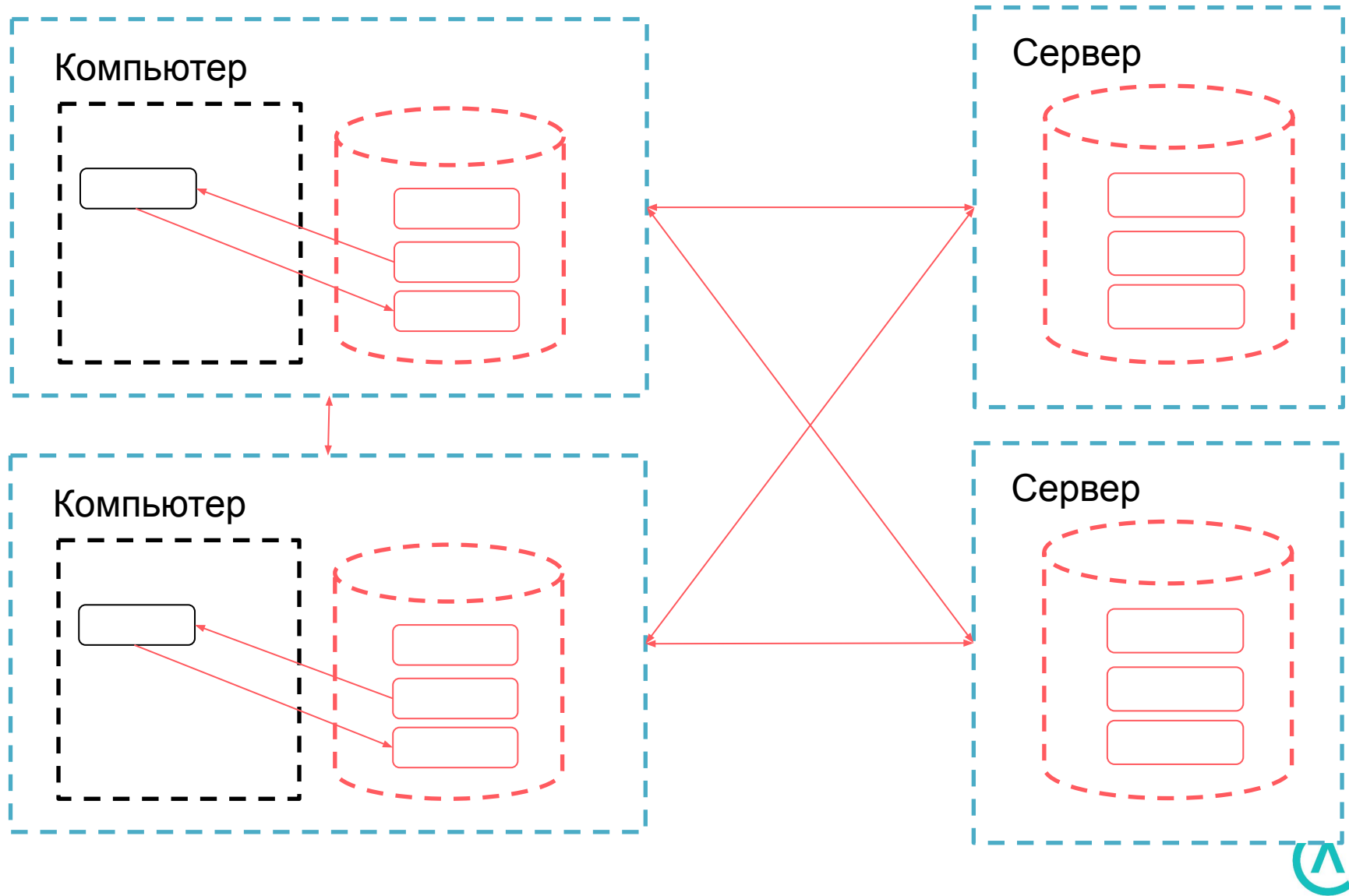
У этой схемы также есть минусы:

1. Нужен отдельный сервер
2. Если с сервером что-то случится, то вся история проекта будет утеряна (но на компьютерах пользователей останется хотя бы текущая версия проекта)
3. Пользователи должны быть подключены к серверу (иначе они не смогут сделать ни commit, ни checkout)

Плюсы, конечно же, в том, что на сервере, фактически, хранится резервная копия.



# Decentralized VCS



# Decentralized VCS

В случае децентрализованной VCS копии репозитория хранятся на отдельных компьютерах (это может быть как пользовательский компьютер, так и отдельный сервер). Каждая копия содержит полную историю изменений. Поэтому пока копия репозитория существует хотя бы у одного пользователя (или на сервере) - есть доступ ко всей истории проекта.



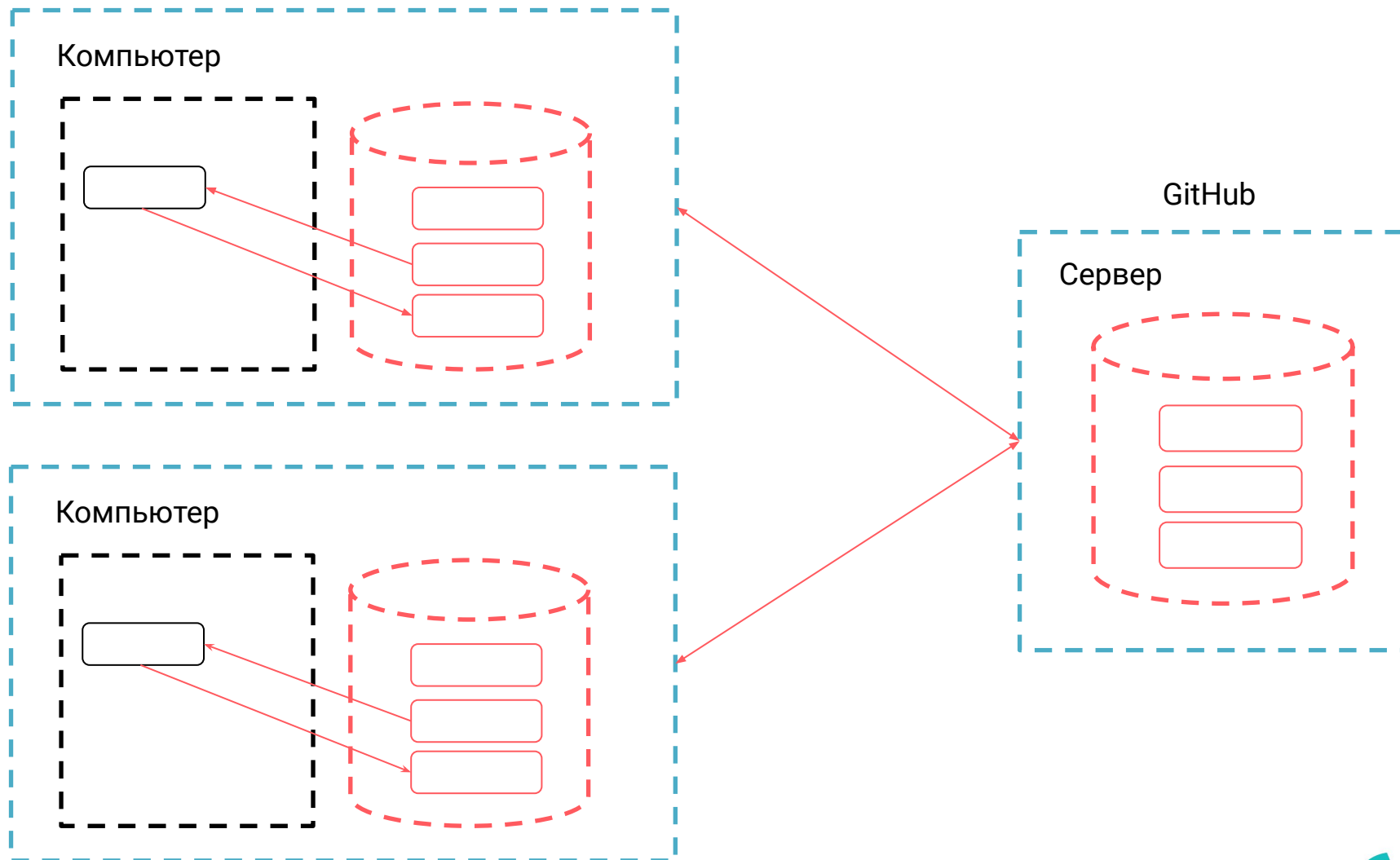
# Decentralized VCS

При этом поскольку копия хранится у каждого пользователя, он вполне может работать без связи с другими участниками, делая коммиты (операция commit) в свою копию.

Единственный существенный минус подобной системы - необходимость синхронизации. Поскольку у каждого своя копия и он может вносить туда изменения без связи с остальными пользователями, ему нужно синхронизировать свою копию с копиями других пользователей. Не всегда это простой процесс, но с этим приходится мириться.



# Самый частый сценарий



# Decentralized VCS

В большинстве случаев используют гибридную схему, когда не каждый пользователь должен синхронизироваться с каждым, а есть централизованный сервер, через который и выполняется синхронизация (при этом на компьютере каждого пользователя по-прежнему хранится полная копия).



# Особенности Git

**Важно:** Git хранит практически полную копию репозитория на локальном компьютере, т.е. у каждого участника есть полная история разработки.

Это позволяет работать без связи с сервером (та самая мечта фрилансеров – работать где-то на пляже под пальмой, даже если нет подключения к сети Интернет).





# Идентификация

Для того, чтобы работать совместно, нужно отличать действия, сделанные одним пользователем, от изменений, сделанных другим.

Git идентифицирует пользователя по двум параметрам:

1. Имя пользователя
2. Email



# Git

Настройки в рамках Git можно осуществлять на трёх уровнях:

- системные (на уровне всего компьютера)
- глобальные (в рамках текущей учётной записи - пользователь)
- локальные (для конкретного репозитория)

Мы с вами будем устанавливать глобальные, чтобы ваш пользователь Windows для всех репозиториев назывался одинаково (и не было необходимости делать эти настройки для каждого репозитория отдельно).



# Git

**Важно:** если вы работаете не на своём персональном компьютере, а на компьютере в общем пространстве, то не устанавливайте на следующем слайде свои данные (ни email, ни имя).



# Git

В терминале VS Code выполните следующие команды (их нужно выполнить только один раз, а не в каждом проекте):

```
git config --global user.name "Student"
```

```
git config --global user.email "student@alif-skills.pro"
```

```
C:\projects\intro> git config --global user.name "Student"
```

```
C:\projects\intro> git config --global user.emai "student@alif-skills.pro"
```

```
C:\projects\intro> █
```

Естественно, вместо Student и [student@alif-skills.pro](mailto:student@alif-skills.pro) вы можете использовать собственные значения (ведь у вас есть email).



# Git

Посмотреть, что вы установили всё правильно, можно с помощью команды **git config --list**:

```
C:\projects\intro>git config --list
pack.packsizelimit=2g
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw32/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
core.editor=nano.exe
pull.rebase=false
credential.helper=manager-core
user.name=Student
user.email=student@alif-skills.pro
```



# Создание репозитория

Создание репозитория выполняется командой `git init`:

```
C:\projects\intro>git init  
Initialized empty Git repository in C:/projects/intro/.git/
```

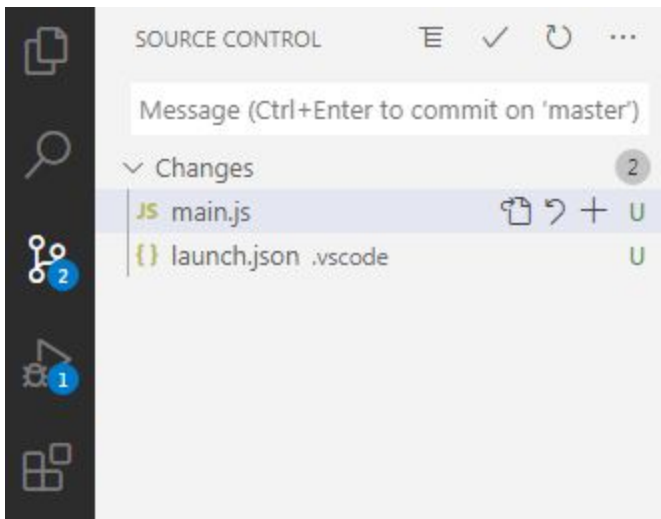
В результате выполнения этой команды появится каталог `.git`, в котором и будет храниться репозиторий и конфигурационные файлы.

Что это значит, это значит, что если вы вдруг удалите каталог с проектом, то каталог `.git` (это и есть ваша копия репозитория) тоже удалится.



# VS Code

В VS Code уже есть готовая интеграция с Git (но мы будем делать всё через консоль, поскольку это наиболее правильный путь):



# **GIT WORKFLOW**





# Рабочий процесс

После того, как вы инициализировали пустой репозиторий, рабочий процесс (workflow) выглядит следующим образом:

1. Делаем изменения в рабочем каталоге (добавляем, удаляем, редактируем файлы)
2. Добавляем изменения в набор для фиксации
3. Производим фиксацию
4. Отправляем на удалённый сервер



# Рабочий процесс

Ключевые моменты:

1. Git отслеживает только те файлы, которые мы ему укажем (нужно их добавить в отслеживаемые)
2. Git фиксирует состояния только тогда, когда мы это явно укажем (нужно явно зафиксировать состояния файлов)
3. Git отслеживает только файлы (т.е. если в каталоге нет отслеживаемых файлов, то каталог не будет учитываться в истории)



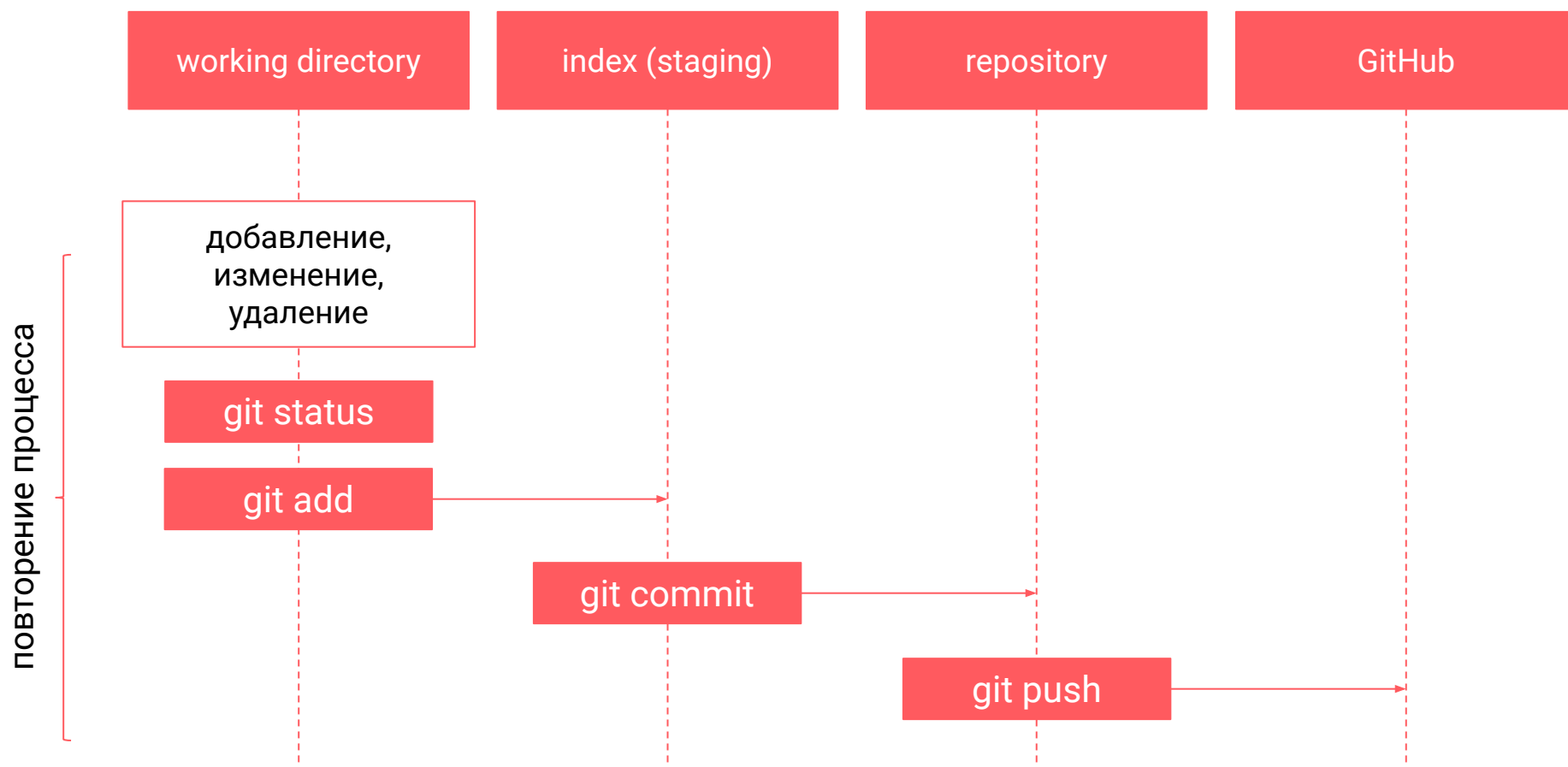
# Рабочий процесс

У всех файлов есть всего 5 состояний:

1. файл игнорируется
2. файл не отслеживается
3. modified (файл добавлен/удалён/изменён)
4. staged (изменения добавлены в список для фиксации)
5. committed (изменения зафиксированы)



# Рабочий процесс



# Рабочий процесс

просмотр статуса:

```
git status
```

добавляем все неотслеживаемые файлы в отслеживаемые и изменения в индекс:

```
git add .
```

или если хотим по одному:

```
git add main.js
```

можно целым каталогом:

```
git add .vscode
```



# Рабочий процесс

просмотр статуса:

```
git status
```

фиксация всех добавленных изменений:

```
git commit -m "v1, список и добавление постов"
```

просмотр статуса:

```
git status
```

просмотр истории

```
git log --oneline
```

после чего повторяем процесс, начиная с команд предыдущего слайда.



```
C:\projects\posts>git init
Initialized empty Git repository in C:/projects/posts/.git/
```

```
C:\projects\posts>git status
On branch master
```

Initial commit

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
    .vscode/
    main.js
```

nothing added to commit but untracked files present (use "git add" to track)

```
C:\projects\posts>git add .
```

```
C:\projects\posts>git status
On branch master
```

Initial commit

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
```

```
new file:   .vscode/launch.json
new file:   main.js
```

```
C:\projects\posts>git commit -m "v1, список и добавление постов"
[master (root-commit) ab8b065] v1, список и добавление постов
 2 files changed, 76 insertions(+)
 create mode 100644 .vscode/launch.json
 create mode 100644 main.js
```

```
C:\projects\posts>git status
On branch master
nothing to commit, working directory clean
```

```
C:\projects\posts>git log --oneline
ab8b065 v1, список и добавление постов
```



# Branch

Вы могли увидеть на скриншоте на предыдущем слайде фразу `branch master` (у вас может быть `main`). Branch (ветка) – это путь, который связывает серию коммитов (т.е. версий):



Git при создании репозитория сам создаёт ветку `master` (или `main`, если вы поменяли значение по умолчанию при установке) и все коммиты, которые мы будем делать, по умолчанию будут выстраиваться "друг за дружкой" в ветке `master`.

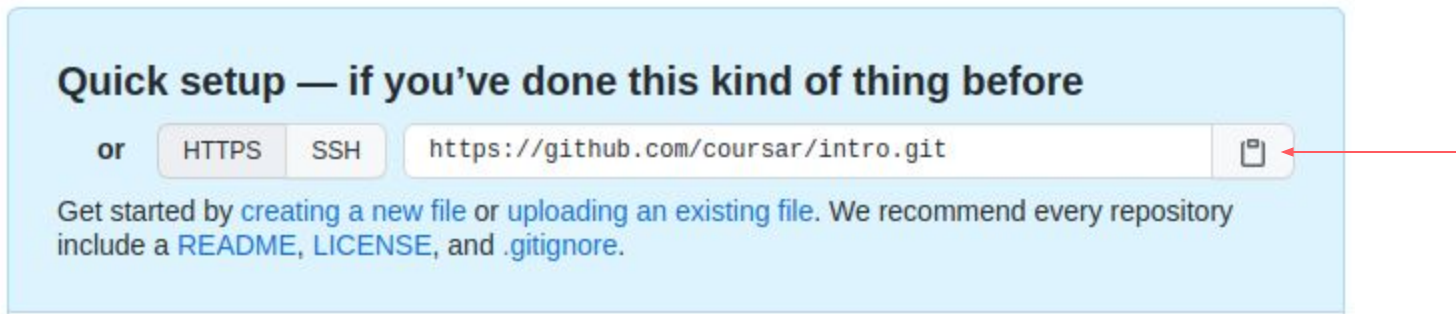
Коммиты выстраиваются в родительско-дочерние отношения (т.е. одна версия порождается из другой), при этом у одного коммита может быть несколько родительских (об этом чуть позже).






# GitHub

Пока наш репозиторий (и все коммиты) хранятся только локально. Это очень плохо. Поэтому перейдите на GitHub и скопируйте ссылку на созданный вами репозиторий:



**Quick setup — if you've done this kind of thing before**

or **HTTPS** **SSH** `https://github.com/coursar/intro.git` 

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

A red arrow points from the right side of the image to the copy icon.



# GitHub

Чтобы "привязать" удалённый репозиторий (вы его создали после регистрации на GitHub), выполните следующую команду:

```
git remote add origin https://github.com/coursar/intro.git
```

Вместо ссылки подставляете вашу ссылку.

Если что-то пошло не так, вы всегда можете выполнить команду `git remote rm origin` (удаление привязки) и повторить всё заново.



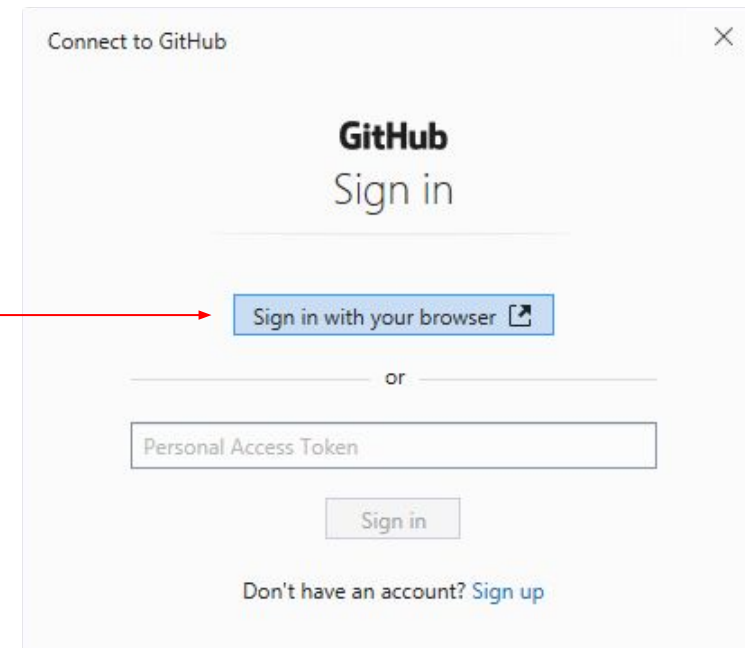
# GitHub

Чтобы отправить локальные коммиты (те, что есть только на вашем компьютере) в удалённый репозиторий\*:

`git push --all`

После этого откроется окно Git Credential Manager, в котором необходимо нажать на кнопку

"Sign in with your browser"



Примечание\*: он называется удалённым потому, что "удалён" (т.е. далеко) от вас, а не потому что, его удалили (т.е. переместили в корзину или стёрли).

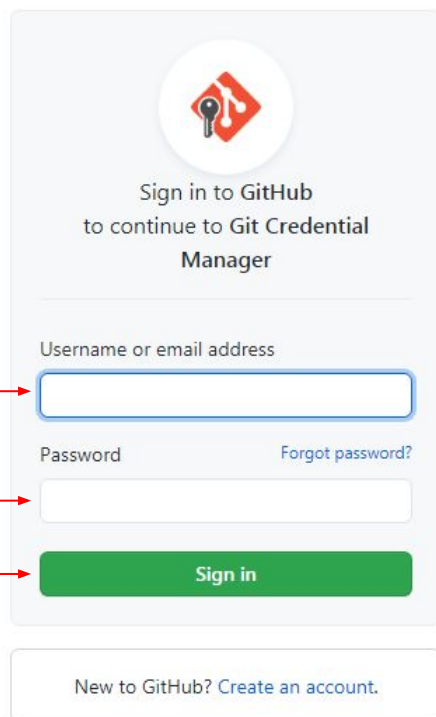


# GitHub

В открывшемся окне браузера введите свои учётные данные GitHub для подтверждения:



Authorize Git Credential Manager



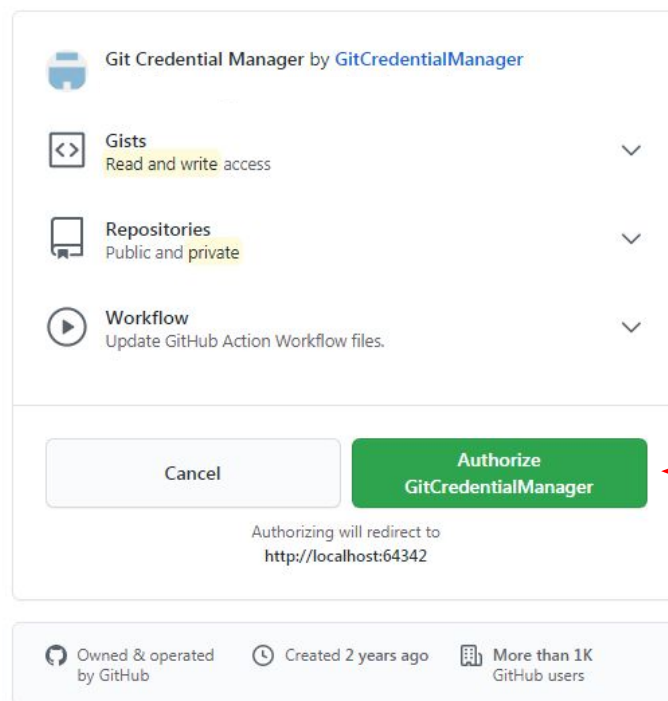
Sign in to GitHub  
to continue to Git Credential  
Manager

Username or email address

Password [Forgot password?](#)

**Sign in**

New to GitHub? [Create an account.](#)



Git Credential Manager by [GitCredentialManager](#)

**Gists**  
Read and write access

**Repositories**  
Public and private

**Workflow**  
Update GitHub Action Workflow files.

**Cancel** **Authorize GitCredentialManager**




Authorizing will redirect to  
<http://localhost:64342>

Owned & operated by GitHub | Created 2 years ago | More than 1K GitHub users





# GitHub



После этого все ваши коммиты (и файлы) появятся в удалённом репозитории:

 master ▾  1 branch  0 tags

[Go to file](#) [Add file ▾](#) [Code ▾](#)

 **coursar** v1, список и добавление постов

33c4d2a 28 seconds ago  1 commit

 .vscode	v1, список и добавление постов	28 seconds ago
 main.js	v1, список и добавление постов	28 seconds ago

Help people interested in this repository understand your project by adding a README.

[Add a README](#)



# Рефакторинг

После того, как мы сохранили и зафиксировали в истории рабочую версию, мы можем уже улучшать код.

Это действительно важно: очень часто в процессе улучшения всё "ломается" и если нет быстрого и безопасного способа вернуть всё "как было", вы просто потратите кучу времени.



# РЕФАКТОРИНГ



# Рефакторинг

Первое, с чего мы начнём, это деструктуризация (destructuring).

Деструктуризация – это возможность извлекать из объектов нужные поля в сокращённом виде.

Например, вместо того чтобы писать:

```
const url = new URL(...);
```

```
const pathname = url.pathname;           —→ const {pathname, searchParams} = new URL(...);
```

```
const searchParams = url.searchParams;
```





# Array Destructuring

Мы специально будем использовать **let**, чтобы не придумывать новые имена:

```
const items = [1, 2, 3];
```

```
let [first, second] = items; // first = 1, second = 2
```

```
let [, first, second] = items; // first = 2, second = 3
```

```
let [first, ...second] = items; // first = 1, second = [2, 3]
```

```
[a, b] = [b, a]; // поменять две переменные значениями
```

```
[items[0], items[2]] = [items[2], items[0]]; // "переставить" два элемента в массиве
```



# Array Destructuring

С функциями работает всё так же:

```
function demo() {  
    return [1, 2]  
}  
  
const [a, b] = demo();
```



# Object Destructuring

То, что нам понадобится больше всего:

```
const item = { a: 10, b: 20, c: 30};
```

```
let {a, b} = item; // a = 10, b = 20
```

```
let {a, ...rest} = item; // a = 10, rest = { b: 20, c: 30 }
```

```
let {a: nA, b: nB} = item; // nA = 10, nB = 20 (as syntax) - с переименованием
```

```
{a, d = 'default'} = item; // default values - если значения нет, назначаем d значение по умолчанию ('default')
```



# Object Destructuring

Можно использовать и для параметров функции:

```
function demo({id}) {
```

```
  ...
```

```
}
```

```
demo({id: 1, name: 'Vasya'}); // в функции id будет равно 1
```



# Используем

```
44 const server = http.createServer(function(request, response) {
45   const {pathname, searchParams} = new URL(request.url, `http://${request.headers.host}`);
46
47   const method = methods.get(pathname);
48   if (method === undefined) {
49     response.writeHead(statusNotFound);
50     response.end();
51     return;
52   }
53
54   const params = {
55     request, // сокращённая запись, аналог request: request (имя свойства = имени переменной)
56     response,
57     pathname,
58     searchParams,
59   };
60
61   method(params);
62 });
```

Обратите внимание, мы всё собрали в объект (чтобы потом деконструировать только нужные свойства в функции).



# Git

```
44  const server = http.createServer(function(request, response) {  
45      const {pathname, searchParams} = new URL(request.url, `http://${request.headers.host}`);  
46  
47      const method = methods.get(pathname);  
48      if (method === undefined) {  
49          response.writeHead(statusNotFound);  
50          response.end();  
51          return;  
52      }  
53  
54      const params = {  
55          request, // сокращённая запись, аналог request: request (имя свойства = имени переменной)  
56          response,  
57          pathname,  
58          searchParams,  
59      };  
60  
61      method(params);  
62  });
```

VS Code отмечает на полях те строки, которые вы изменили по сравнению с той версией, что зафиксировали последней в Git.



# Git

```
54  ✓  const params = {
55      request, // сокращённая запись, аналог request: request (имя свойства = имени переменной)
56      response,
57      pathname,
58      searchParams,
59  };
60
61  method(params);
```

---

main.js 2 of 2 changes

52	51	return;
53	52	}
54	53	
55	-	method(request, response);
54+		const params = {
55+		request, // сокращённая запись, аналог request: request (имя свойства = имени переменной)
56+		response,
57+		pathname,
58+		searchParams,
59+		};
60+		
61+		method(params);

Можно кликнуть и посмотреть, что изменилось: **красное** – изменили (удалили), **зелёное** – добавили. Git не понимает слов "изменили строку". Он всегда считает, что вы удалили одну и добавили другую строку.



# Destructuring

Обратите внимание: при деструктуризации мы берём только те свойства, которые нам нужны.

Если никакие пока не нужны, то оставляем пустые скобки.

```
13  const methods = new Map();
14  methods.set('/posts.get', function({response}) {
15      response.writeHead(statusOk, {'Content-Type': 'application/json'});
16      response.end(JSON.stringify(posts));
17  });
18  methods.set('/posts.getById', function() {});
19  methods.set('/posts.post', function({response, searchParams}) {
20      if (!searchParams.has('content')) {
21          response.writeHead(statusBadRequest);
22          response.end();
23          return;
24      }
25
26      const content = searchParams.get('content');
27
28      const post = {
29          id: nextId++,
30          content: content,
31          created: Date.now(),
32      };
33
34      posts.unshift(post);
35      response.writeHead(statusOk, {'Content-Type': 'application/json'});
36      response.end(JSON.stringify(post));
37  });
38  methods.set('/posts.edit', function() {});
39  methods.set('/posts.delete', function() {});
```





# Commit

После сделанных изменений нужно проверить, что всё работает и сделать коммит:

```
C:\projects\posts>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   main.js

no changes added to commit (use "git add" and/or "git commit -a")

C:\projects\posts>git add .

C:\projects\posts>git commit -m "refactor: деструктуризация в аргументах обработчиков"
[master 818a0a9] refactor: деструктуризация в аргументах обработчиков
 1 file changed, 15 insertions(+), 12 deletions(-)

C:\projects\posts>git status
On branch master
nothing to commit, working directory clean

C:\projects\posts>git log --oneline
818a0a9 refactor: деструктуризация в аргументах обработчиков
ab8b065 v1, список и добавление постов
```



# Функции

Следующим этапом напрашивается написание вспомогательных функций, которые бы отправляли ответ (вместе с заголовками).

Давайте попробуем написать такую универсальную функцию (или несколько).



# Функции

```
13 function sendResponse(response, {status = statusOk, headers = {}, body = null}) {
14     Object.entries(headers).forEach(function([key, value]) {
15         response.setHeader(key, value);
16     });
17     response.writeHead(status);
18     response.end(body);
19 }
20
21 function sendJSON(response, body) {
22     sendResponse(response, {
23         headers: {
24             'Content-Type': 'application/json',
25         },
26         body: JSON.stringify(body),
27     });
28 }
29
```

Как вы видите, мы написали целых две функции:

1. Общую, в которой можно указать все параметры (и значения по умолчанию для них)
2. Отправляющую JSON (при этом код будет 200 и произведены необходимые действия: выставлены заголовки и `JSON.stringify`)



```
30 const methods = new Map();
31 methods.set('/posts.get', function({response}) {
32     |   sendJSON(response, posts);
33 });
34 methods.set('/posts.getById', function() {});
35 methods.set('/posts.post', function({response, searchParams}) {
36     |   if (!searchParams.has('content')) {
37     |       |   sendResponse(response, {status: statusBadRequest});
38     |       |   return;
39     |   }
40
41     |   const content = searchParams.get('content');
42
43 >   const post = { ...
47     |   };
48
49     |   posts.unshift(post);
50     |   sendJSON(response, post);
51 });
52 methods.set('/posts.edit', function() {});
53 methods.set('/posts.delete', function() {});
54
55 const server = http.createServer(function(request, response) {
56     |   const {pathname, searchParams} = new URL(request.url, `http://${request.headers.host}`);
57
58     |   const method = methods.get(pathname);
59     |   if (method === undefined) {
60     |       |   sendResponse(response, {status: statusNotFound});
61     |       |   return;
62     |   }
63
64 >   const params = { ...
69     |   };
70
71     |   method(params);
72 });
```



# Commit

```
C:\projects\posts>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
    modified:   main.js
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

```
C:\projects\posts>git add .
```

```
C:\projects\posts>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
    modified:   main.js
```

```
C:\projects\posts>git commit -m "refactor: вспомогательные функции sendResponse & sendJSON для отправки ответа"
[master 6899892] refactor: вспомогательные функции sendResponse & sendJSON для отправки ответа
1 file changed, 21 insertions(+), 8 deletions(-)
```

```
C:\projects\posts>git status
On branch master
nothing to commit, working directory clean
```

```
C:\projects\posts>git log --oneline
6899892 refactor: вспомогательные функции sendResponse & sendJSON для отправки ответа
818a0a9 refactor: дефрагментация в аргументах обработчиков
ab8b065 v1, список и добавление постов
```



# Рефакторинг

Обратите внимание: мы создали функции только после того как код начал дублироваться. Пока он не дублируется – не нужно создавать лишних функций и всего прочего.



# БАЗЫ ДАННЫХ



# Базы данных

Хранить данные в памяти, конечно, хорошо. Но в промышленных масштабах – уже не очень (представьте, в Vk бы стёрлись все данные, если бы перезагрузился сервер).

Мы можем хранить всё в файлах (и вы даже знаете как – JSON можно очень легко записать в файл, а потом прочитать оттуда), но файлы – это не очень удобно.





# Базы данных

Базы данных – это отдельные программные продукты, которые предоставляют нам богатый набор функциональности:

1. Структура информации
2. Запросы
3. Целостность
4. Транзакции
5. Журналирование
6. Многопользовательский доступ

С ними (базами данных), а также языком SQL мы познакомимся в рамках следующей лекции.



# ИТОГИ



# ИТОГИ

В этой лекции мы попробовали реализовать простенький сервер с API, похожим на API Vk (это, конечно, очень громко сказано, но мы ориентировались на них).

Дальше мы продолжим работать в этом направлении и посмотрим на хранение данных в базах данных и на варианты другой организации API.



# ДОМАШНЕЕ ЗАДАНИЕ



# Как сдавать ДЗ

В рамках этой лекции вы должны сделать один большой проект, в рамках которого нужно реализовать все требования (бот будет проверять реализацию всех требований ДЗ).



# ДЗ №1: getByld

Напишите реализацию функции `getByld`, которая будет возвращать пост по его `id`.

Бот будет присылать запрос следующего вида:

<http://localhost/posts.getByld?id=1> (не обязательно 1).

Вы должны:

1. Отдавать код 200 и ответ в виде JSON (не забудьте заголовок), если пост найден
2. Отдавать код 400, если пользователь прислал "плохой" запрос (см. следующий слайд)
3. Отдавать код 404, если пост с таким `id` не найден



# ДЗ №1: getById

Поскольку это клиент присылает вам запрос (другое приложение - мы просто тестируем через браузер), то он может прислать плохой запрос:

1. Не указать id
2. Указать id неправильно, например, id=post (т.е. не число, а строка)

Вы должны обрабатывать эти ситуации и возвращать код 400.



# ДЗ №1: getById

Обратите внимание, что всё, что приходит в строке запроса – это строка. А id у нас – это число. Поэтому используйте функцию [Number](#) (в виде функции, а не в виде конструктора, чтобы преобразовать строку к числу). При преобразовании, вы можете получить значение [NaN](#) (Not a Number – число, которое не число)

Это значение примечательно тем, что оно ничему не равно, включая самого себя.

Т.е.:

```
> NaN === NaN  
< false  
> |
```

Поэтому проверять на то, что вы получили **NaN** нужно с помощью функции [Number.isNaN\(\)](#).





## ДЗ №2: edit

Напишите реализацию функции edit, которая будет обновлять пост (т.е. изменять свойство **content**).

Бот будет присылать запрос следующего вида:

<http://localhost/posts.edit?id=1&content=Updated> (необязательно **1** и **Updated**).

Вы должны:

1. Отдавать код 200 и ответ в виде JSON (обновлённый пост), если пост найден
2. Отдавать код 400, если пользователь прислал "плохой" запрос (нет id, id – не число или нет content)
3. Отдавать код 404, если пост с таким id не найден



# ДЗ №3: delete

Напишите реализацию функции `delete`, которая будет удалять пост (см. методы [findIndex](#), [splice](#)).

Бот будет присылать запрос следующего вида:

<http://localhost/posts.delete?id=1> (не обязательно 1).

Вы должны:

1. Отдавать код 200 и ответ в виде JSON (удалённый пост), если пост найден
2. Отдавать код 400, если пользователь прислал "плохой" запрос (нет id, id - не число)
3. Отдавать код 404, если пост с таким id не найден



# ДЗ №4: safe delete

Самое худшее, что можно сделать с данными – это удалить их. Пользователи всегда совершают ошибки "случайно" удаляя данные (чаще всего они потом уверяют вас, что ничего не делали).

Поэтому чаще всего данные не удаляются, а просто помечаются, как удалённые. Т.е. вам нужно добавить в каждый пост при создании поле **removed** (типа **boolean**), в котором записывать **false**, если пост не удалён и **true**, если удалён.

При этом, если пост помечен как удалённый, то **get** не должен возвращать его в общем массиве (посмотрите на функцию [filter](#)), а **getById**, **edit**, **delete** должны возвращать 404.



## ДЗ №5: restore

Вам нужно сделать функцию `posts.restore` (аналог Vк [wall.restore](#)), которая "восстанавливает" пост (фактически просто меняет `removed` на `false`). Делать это нужно только тогда, когда ваш сервер уже умеет "безопасно" удалять посты (т.е. вы сделали ДЗ №4).

Бот будет отправлять такой запрос:

<http://localhost:9999/posts.restore?id=1> (не обязательно 1)

Вы должны:

1. Отдавать код 200 и ответ в виде JSON (обновлённый пост), если пост найден и был удалён
2. Отдавать код 400, если пользователь прислал "плохой" запрос (нет id, id – не число или пост не был удалён)
3. Отдавать код 404, если пост с таким id не найден



Спасибо за внимание

**alif skills**

2023г.

