

Toolkit

CDN

CDN

CDN (Content Delivery Network) — это географически распределённая сетевая инфраструктура, обеспечивающая быструю доставку контента пользователям веб-сервисов и сайтов. Входящие в состав CDN серверы географически располагаются таким образом, чтобы сделать время ответа для пользователей сайта/сервиса минимальным.

PoP (point of presence, точка присутствия) — кэширующий сервер в составе CDN, расположенный в определенной географической локации. Для обозначения таких серверов также используется термин edge.

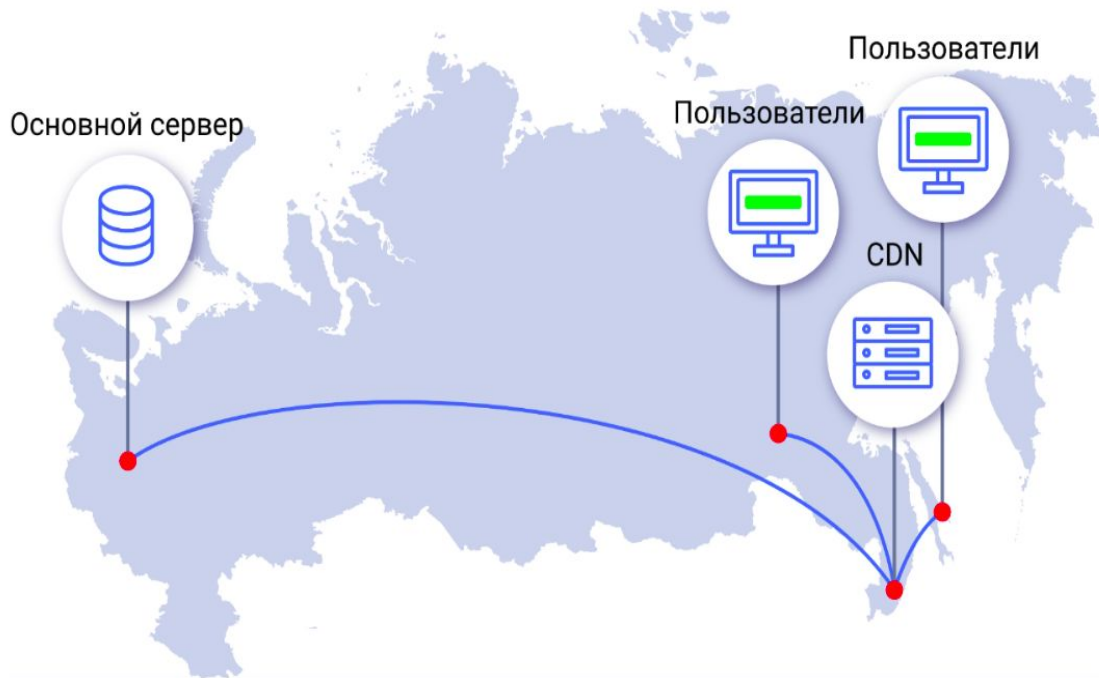
Динамический контент — контент, генерируемый на сервере в момент получения запроса (либо изменяемый пользователем, либо загружаемый из базы данных).

Статический контент — контент, хранимый на сервере в неизменяемом виде (например, бинарные файлы, аудио- и видеофайлы, JS и CSS).

CDN

Проблема:

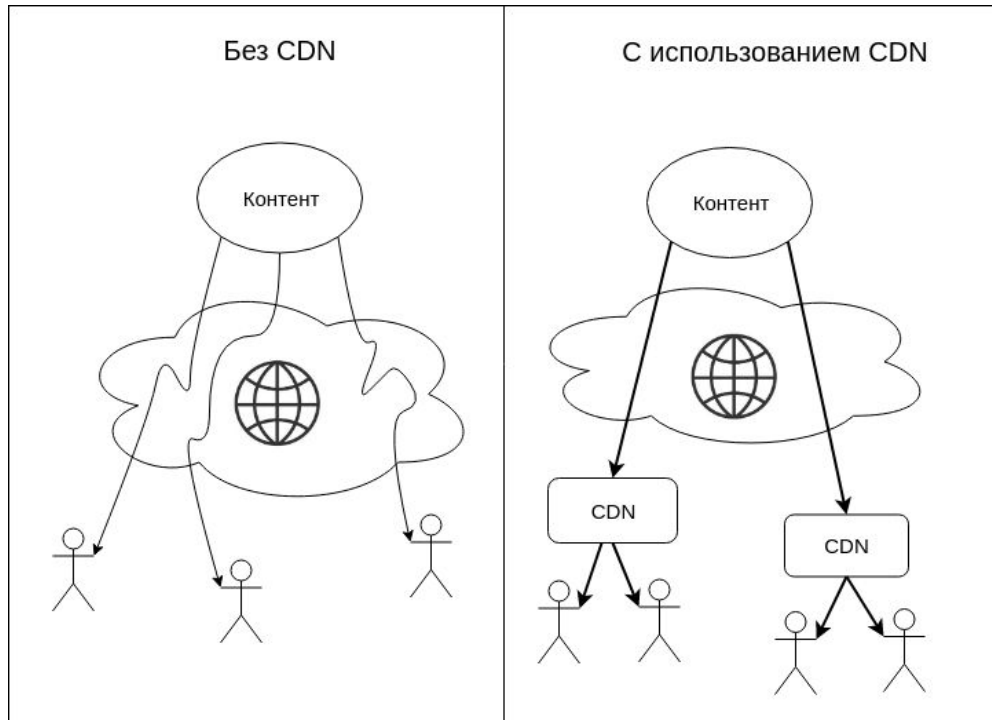
Чем дальше пользователь находится от оригинального сервера, тем больше время «оригинального» ответа.



CDN

Решение - Content Delivery Network

Пользователь переадресуется к географически ближайшему кэширующему серверу в составе CDN.



CDN - как кэшируется контент?

Самой распространенной является схема по первому обращению: максимальное количество времени на загрузку затрачивает пользователь, обратившийся к оригинальному серверу первым. Все последующие пользователи будут получать данные, кэшированные на ближайшей к ним точке присутствия.

CDN

Чтобы подключить библиотеку с использованием cdn, достаточно использовать тег script с указанием соответствующего адреса в атрибуте src или воспользоваться картой импортов.

В качестве - библиотека three.js:

```
<script async src="https://unpkg.com/es-module-shims@1.3.6/dist/es-module-shims.js"></script>
<script type="importmap">
  {
    "imports": {
      "three": "https://unpkg.com/three@<version>/build/three.module.js"
    }
  }
</script>
<script type="module">
  import * as THREE from 'three';

  const scene = new THREE.Scene();
</script>
```

Подгрузка шимов

Описание модуля three в карте импортов

Использование модуля

NPM

NPM

npm (Node Package Manager) – дефолтный пакетный менеджер для JavaScript, работающий на Node.js. Он состоит из 2 частей:

- CLI (интерфейс командной строки) – средство для размещения и скачивания пакетов,
- онлайн-репозитории, содержащие JS пакеты.



npm install



NPM - установка

NPM поставляется вместе с node.js. Загрузить его можно с сайта nodejs.org из раздела [download](#):

LTS Recommended For Most Users		Current Latest Features
 Windows Installer node-v16.15.1-x64.msi	 macOS Installer node-v16.15.1.pkg	 Source Code node-v16.15.1.tar.gz
Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit / ARM64	
macOS Binary (.tar.gz)	64-bit	ARM64
Linux Binaries (x64)	64-bit	
Linux Binaries (ARM)	ARMv7	ARMv8
Source Code	node-v16.15.1.tar.gz	

NPM - инициализация проекта

Каждый проект может быть представлен как npm-пакет. Каждый такой пакет имеет файл package.json - дескриптор проекта.

Для инициализации проекта используется команда npm init.

Для инициализации с дефолтными параметрами используется флаг -y (--yes)

Выполнение команды npm init -y приведёт к созданию в текущей директории файла package.json со следующим содержимым:

```
{
  "name": "project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Название проекта

Версия проекта

Описание проекта

Входная точка

Список скриптов

Ключевые слова - помогают в поиске через npm search

Автор проекта

Лицензия проекта

NPM - scripts

Свойство `scripts` файла `package.json` содержит набор скриптов для автоматизации сборки проекта.

Он поддерживает ряд встроенных сценариев и предустановленных событий, а также произвольные сценарии.

Для выполнения скрипта используется команда `npm run-script <stage>` или её краткий аналог `npm run <stage>`.

Пример пользовательских скриптов приведён ниже:

```
{
  "scripts": {
    "build": "tsc",
    "format": "prettier --write **/*.ts",
    "format-check": "prettier --check **/*.ts",
    "lint": "eslint src/**/*.ts",
    "pack": "ncc build",
    "test": "jest",
    "all": "npm run build && npm run format ..."
  }
}
```

Сборка

Форматирование кода

Проверка форматирования кода

Линтер

Сборка в 1 файл

Тестирование

Запуск всех скриптов

NPM - зависимости

`dependencies` и `devDependencies` представляют собой словари с именами npm-библиотек (ключ) и их семантические версии (значение).

Эти зависимости устанавливаются командой `npm install` с флагами `--save` и `--save-dev`. Они предназначены соответственно для использования в продакшене и разработке.

Краткий синтаксис: `npm i`, `npm add`.

Вызов без аргументов приведет к загрузке указанных в `package.json` зависимостей.

Для загрузки конкретного пакета в качестве аргумента передается его имя с возможным указанием конкретной версии или диапазона версий. Загруженные зависимости располагаются в папке `node_modules`

О версионировании:

`^`: последний минорный релиз.

`~`: последний патч-релиз.

```
{
  "dependencies": {
    "@actions/core": "^1.2.3",
    "@actions/github": "^2.1.1"
  },
  "devDependencies": {
    "@types/node": "^13.9.0",
    "typescript": "^3.8.3"
  }
}
```

NPM - package-lock.json

Файл `package-lock.json` описывает версии пакетов, используемые в JavaScript-проекте. Если `package.json` включает общее описание зависимостей, то `package-lock.json` более детальный – всё дерево зависимостей (включая зависимости зависимостей и т.д.).

Каждая зависимость может иметь свои зависимости, которые могут иметь еще больше зависимостей, среди которых часто встречаются дубли:



Оптимизировать дерево зависимостей поможет команда `npm dedupe`.

Действие дедупликации призвано упростить структуру дерева зависимостей путем поиска общих пакетов между ними и их перемещением для последующего переиспользования.

Webpack

webpack - что и зачем

Вебпак — это сборщик статических модулей. Он анализирует модули приложения, создает граф зависимостей, затем собирает модули в правильном порядке в один или более бандл (bundle)

Возможности:

- сборка модулей;
- транспиляция;
- конвертация;
- сервер для разработки;
- three shaking.

Также используется для backend разработки на Node.js.

webpack - основные концепции

- Entry: точка входа - это модуль, который webpack использует для начала построения своего внутреннего графа зависимостей. Значение по умолчанию - `./src/index.js`.
- Output: выходная точка - папка, в которой сохраняются результирующие файлы и их имена. По умолчанию `./dist/main.js` для основного пакета `./dist` для остальных файлов.
- Loaders: по умолчанию webpack работает только с JavaScript и JSON. Лоадеры необходимы для обработки файлов, отличных от JS и JSON.
- Plugins: плагины отвечают за выполнение любых задач отличных от загрузки задач.
- Mode: режим позволяет настроить разные профили сборки: `development`, `production`, `none`.

webpack - начало работы

Рассмотрим практический пример. Для этого необходимо создать новый каталог проекта и инициализировать его:

```
mkdir learn-webpack  
cd learn-webpack  
npm init -y
```

Далее необходимо установить webpack и webpack CLI (command line interface):

```
npm install webpack webpack-cli --save-dev
```

webpack - начало работы

После всех проведенных действий файл `package.json` должен выглядеть следующим образом:

```
{
  "name": "learn-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^5.9.0",
    "webpack-cli": "^4.2.0"
  }
}
```

Дефолтные настройки

Webpack в зависимостях
для разработки

webpack - начало работы

Теперь нужно настроить команды для запуска необходимых задач. Сделать это можно в секции scripts файла package.json:

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "dev": "webpack --mode development",  
  "build": "webpack --mode production"  
},
```

test - дефолтная команда

dev - запуск webpack в режиме development

build - запуск webpack в режиме production

webpack - начало работы

Теперь необходимо создать в корне проекта каталог src, а в нем входную точку по умолчанию - файл index.js:

```
console.log("Hello, Webpack!");
```

Осталось запустить задачу npm run dev, что приведёт к сборке проекта в режиме разработки:

```
$ npm run dev

> learn-webpack@1.0.0 dev C:\WEBDEV\learn-webpack
> webpack --mode development

[webpack-cli] Compilation finished
asset main.js 874 bytes [emitted] (name: main)
./src/index.js 31 bytes [built] [code generated]
webpack 5.9.0 compiled successfully in 122 ms
```

webpack - начало работы

Результат выполнения предыдущей команды - генерация файла main.js в директории по умолчанию dist. Результирующий файл можно указать в файле HTML:

```
<!doctype html>
<html>
  <head>
    <title>Something title</title>
  </head>
  <body>
    <script src="main.js"></script>
  </body>
</html>
```

webpack - файл конфигурации

Начиная с версии 4 и выше, webpack предоставляет настройки по умолчанию из коробки. Предыдущий пример достаточно прост, чтобы можно было обойтись без настройки сборщика модулей. Однако в реальных проектах перед Вами будут стоять немного более трудные задачи, и дефолтного функционала вряд ли хватит. Для тонкой настройки и наращивания возможностей существует конфигурационный файл `webpack.config.js`.

Файл конфигурации не создается по-умолчанию. Его необходимо создавать вручную и следует расположить в корневой папке проекта:

```
module.exports = {  
  /*config*/  
}
```

При работе с файлом конфигурации Вы можете использовать модули `node.js`, например, `path`.

webpack - файл конфигурации

Рассмотрим основные параметры файла конфигурации:

- entry - точка входа. Значение по умолчанию - ./src/index.js

```
module.exports = {  
  entry: './index.js'  
}
```

- output - выходная точка. Значение по умолчанию - ./dist/main.js

```
module.exports = {  
  output: {  
    path: path.resolve(__dirname, 'dist'), // выходная директория  
    filename: 'app.js' // имя выходного файла  
  }  
}
```


Webpack Plugins

webpack - плагины

Теперь рассмотрим работу с плагинами на примере html-webpack-plugin.

Плагины устанавливаются как и любой другой пакет - через npm:

```
npm install html-webpack-plugin@next --save-dev
```

Просто установить плагин - мало. Мы должны объявить его в файле конфигурации в массиве plugins:

```
const HtmlWebpackPlugin = require("html-webpack-plugin"); // Импорт плагина
const path = require('path');

module.exports = {
  plugins: [ // Массив плагинов
    new HtmlWebpackPlugin({ // Добавляется как обычный объект
      title: "Something title", // Настройки плагина
    }),
  ],
};
```

webpack - плагины

Теперь можно удостовериться в работоспособности плагина:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Something title</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script defer src="main.js"></script>
  </head>
  <body>
  </body>
</html>
```

```
$ npm run dev
```

```
> learn-webpack@1.0.0 dev C:\WEBDEV\learn-webpack
> webpack --mode development
```

```
[webpack-cli] Compilation finished
asset main.js 874 bytes [compared for emit] (name: main)
asset index.html 234 bytes [emitted]
./src/index.js 31 bytes [built] [code generated]
webpack 5.9.0 compiled successfully in 151 ms
```

Сейчас выполнение инструкции `npm run dev` приводит к генерации файла `html`, содержащего все необходимые импорты

webpack - entry points

Приведённая ранее конфигурация входной и выходной точек может быть модифицирована:

```
entry: {  
  main: path.resolve(__dirname, './src/app.js'),  
},  
output: {  
  filename: '[name].bundle.js',  
  path: path.resolve(__dirname, 'deploy')  
},
```

entry может являться объектом, описывающим множество входных точек, именами которых являются ключи.

Имена входных точек, как и другая мета-информация, также могут быть использованы в конфигурации. Например, можно задать имена выходных файлов в зависимости от имён входных.

webpack - entry points

До текущего момента рассматриваемое приложение состояло из одного js файла. Пора создать ещё один - /src/component.js:

```
export default (text = "Hello, Webpack!") => {  
  const element = document.createElement("h1");  
  
  element.innerHTML = text;  
  
  return element;  
};
```

Теперь необходимо переименовать главный файл index.js в app.js для соответствия конфигурации и подключить к нему созданный выше модуль:

```
import component from './component';  
  
document.body.appendChild(component());
```

webpack - entry points

Теперь необходимо запустить сборку модифицированного проекта:

```
$ npm run dev

> learn-webpack@1.0.0 dev C:\WEBDEV\learn-webpack
> webpack --mode development

[webpack-cli] Compilation finished
asset main.bundle.js 4.67 KiB [emitted] (name: main)
asset index.html 241 bytes [emitted]
runtime modules 668 bytes 3 modules
cacheable modules 230 bytes
  ./src/app.js 79 bytes [built] [code generated]
  ./src/component.js 151 bytes [built] [code generated]
webpack 5.9.0 compiled successfully in 194 ms
```

В процессе сборки был обработан главный файл, а также файл созданного компонента.

Результатом сборки стали файлы main.bundle.js и index.html

webpack - транспиляция

Webpack позволяет транспилировать ES6 в ES5-совместимый код, который работает во всех браузерах. Начнем с выполнения следующей команды, которая сделает выходной код более читабельным:

```
npm run dev -- --devtool inline-source-map
```

```
/***/ "./src/component.js":  
/*! *****!\n    !*** ./src/component.js ***!  
    \n*****/  
/*! namespace exports */  
/*! export default [provided] [no usage info] [missing usage info prevents renaming] */  
/*! other exports [not provided] [no usage info] */  
/*! runtime requirements: __webpack_exports__, __webpack_require__.r, __webpack_require__.d,  
__webpack_require__.* */  
/***/ ((__unused_webpack_module, __webpack_exports__, __webpack_require__) => {  
  
    __webpack_require__.r(__webpack_exports__);  
/* harmony export */ __webpack_require__.d(__webpack_exports__, {  
/* harmony export */    "default": () => __WEBPACK_DEFAULT_EXPORT__  
/* harmony export */ });  
/* harmony default export */ const __WEBPACK_DEFAULT_EXPORT__ = ((text = "Hello, Webpack!") => {  
    const element = document.createElement("h1");  
  
    element.innerHTML = text;  
  
    return element;  
});  
/***/ })
```

По умолчанию возможности языка es6 (стрелочные функции, const объявления) не транспилируются в код es5. Чтобы добавить эту возможность, необходимо воспользоваться специальным загрузчиком.

webpack - транспилиция

В качестве транспилятора будет использоваться babel. Для работы с ним необходимо установить следующие зависимости:

```
npm install babel-loader @babel/core @babel/preset-env --save-dev
```

Теперь необходимо добавить загруженные пакеты в конфиг. Загрузчики добавляются в поле module.rules:

```
module: {
  rules: [
    {
      test: /\.js$/,           // Какие файлы должны обрабатываться
      exclude: /node_modules/, // Какие файлы НЕ должны обрабатываться
      use: {                   // Что следует использовать для обработки модуля
        loader: 'babel-loader', // Загрузчик, который следует использовать
        options: {              // Параметры загрузчика
          presets: ['@babel/preset-env'] // Наборы параметров загрузчика
        }                       // (их мы установили вместе с babel)
      }
    },
  ],
},
```


webpack - транспиляция

Теперь, если мы вновь соберём проект в режиме повышенной читабельности, увидим следующие изменения:

```
/***/ "./src/component.js":
/*!*****!\
  !*** ./src/component.js ***!
  \*****\
  /*! namespace exports */
  /*! export default [provided] [no usage info] [missing usage info prevents renaming] */
  /*! other exports [not provided] [no usage info] */
  /*! runtime requirements: __webpack_exports__, __webpack_require__.r, __webpack_require__.d,
  __webpack_require__.* */
  !***/ ((__unused_webpack_module, __webpack_exports__, __webpack_require__) => {
    __webpack_require__.r(__webpack_exports__);
    /* harmony export */ __webpack_require__.d(__webpack_exports__, {
    /* harmony export */    "default": () => __WEBPACK_DEFAULT_EXPORT__
    /* harmony export */ });
    /* harmony default export */ const __WEBPACK_DEFAULT_EXPORT__ = (function () {
      var text = arguments.length > 0 && arguments[0] !== undefined ? arguments[0] : "Hello, Webpack!";
      var element = document.createElement("h1");
      element.innerHTML = text;
      return element;
    });
  });
  !***/ })
```

Теперь мы можем использовать современные функции JS, а webpack преобразует наш код, чтобы его можно было выполнять в старых браузерах.

Webpack Loaders

webpack - работа со стилями

Следующий этап - работа со стилями. Следующие 2 плагина добавляют в функционал сборщика возможность работать с css:

```
npm install css-loader style-loader --save-dev
```

- css-loader - возвращает css-код при импорте css файлов, разрешает @import и url(...);
- style-loader - выводит css в <style> тег в документе html.

Далее необходимо создать в конфиге соответствующие правила. Важен порядок лоадеров в массиве - справа налево. Сначала css анализируется, потом добавляется в html:

```
rules: [  
  ...  
  {  
    test: /\.css$/,  
    use: ["style-loader", "css-loader"]  
  },  
]
```

webpack - работа с ресурсами

Чаще всего ваш проект содержит ресурсы, такие как изображения, шрифты и так далее. В webpack 4 для работы с ресурсами необходимо было установить один или несколько из следующих загрузчиков: file-loader, raw-loader и url-loader. В webpack 5, как мы видели ранее, это больше не требуется, поскольку новая версия поставляется со встроенными модулями ресурсов. Рассмотрим пример с изображениями. Для работы с ними необходимо добавить новое правило в webpack.config.js:

```
rules: [  
  ...  
  {  
    test: /\.?(?:ico|gif|png|jpg|jpeg)$/i,  
    type: 'asset/resource',  
  },  
]
```

Здесь type: 'asset/resource' используется вместо указания ладера (например, file-loader).

webpack - работа с ресурсами

Asset/resource - не единственный тип ресурсов, поддерживаемый webpack:

- `asset/resource` - создает отдельный файл и экспортирует URL. Выполняет функцию `file-loader`.
- `asset/inline` - экспортирует URI данных ресурса. Выполняет функцию `url-loader`.
- `asset/source` - экспортирует исходный код ресурса. Выполняет функцию `raw-loader`.
- `asset` - автоматически выбирает между экспортом URI данных и созданием отдельного файла. Раньше выполнялось с помощью `url-loader` с ограничением размера ресурса.
- `javascript/auto` - позволяет запретить модулю ресурсов обработку файлов. Может быть полезно при использовании внешних лоадеров во избежание дублирования ресурсов.

webpack - работа с ресурсами

По умолчанию модули ресурсов сохраняются в выходной каталог с именем `[hash][ext][query]`.

Стандартное поведение можно изменить, задав `output.assetModuleFilename` в конфигурации webpack:

```
module.exports = {
  output: {
    /*...*/
    assetModuleFilename: 'images/[hash][ext][query]'
  },
};
```

Также можно изменить этот параметр для конкретных правил, добавив блок `generator`:

```
rules: [
  {
    /*...*/
    generator: {
      filename: 'static/[hash][ext][query]'
    }
  }
]
```

webpack - сервер для разработки

Dev-server или сервер для разработки - веб-сервер с перезагрузкой в реальном времени, который автоматически создает и обновляет страницу. Установить его можно через npm:

```
npm install webpack-dev-server --save-dev
```

После установки необходимо настроить сервер в конфигурационном файле:

```
devServer: {  
  static: {  
    directory: './deploy' // Обслуживание статичных файлов  
    // Расположение статичных файлов  
  },  
  open: true // Открыть окно в браузере после запуска сервера  
}
```

Для запуска сервера используется команда webpack с параметром serve:

```
"scripts": {  
  /*...*/  
  "dev": "webpack serve --mode development"  
}
```

webpack - очистка выходного каталога

При каждой сборке проекта в директории deploy становится всё больше файлов - старые сборки не очищаются автоматически. Эту проблему решает плагин clean-webpack-plugin. Устанавливается он как и любой другой плагин - с помощью npm и обязательно --save-dev:

```
npm install clean-webpack-plugin --save-dev
```

Для начала работы достаточно добавить его в список плагинов конфигурационного файла:

```
const { CleanWebpackPlugin } = require('clean-webpack-plugin');

/*...*/
plugins: [
  /*...*/
  new CleanWebpackPlugin()
],
```

Теперь директория deploy будет содержать только файлы, сгенерированные при последней сборке проекта.

Postman

Postman

- **Postman** — это набор инструментов для тестирования API. Он является средой разработки для создания, тестирования, контроля и публикации API-документации.
- **Назначение Postman** — тестирование отправки запросов с клиента на сервер и получения ответа от сервера.



API Tools

A comprehensive set of tools that help accelerate the API Lifecycle - from design, testing, documentation, and mocking to discovery.



API Repository

Easily store, iterate and collaborate around all your API artifacts on one central platform used across teams.



Workspaces

Organize your API work and collaborate with teammates across your organization or stakeholders across the world.



Intelligence

Improve API operations by leveraging advanced features such as search, notifications, alerts and security warnings, reporting, and much more.

HomeWorkspaces▼ReportsExplore

Search Postman

9+InviteTeam▼

Twitter's Public WorkspaceNewImport

Collections

Twitter API v2 / Fork label

Tweet Lookup

GET Single Tweet

200 Success - Request

200 Success - Request

200 Success - Request

200 Success - Request

200 Success - Default

429 Rate Limit Exceed

200 Success - Deleted

GET Single Tweet Usercontext

GET Multiple Tweets

User Lookup

Follows

Blocks

Likes

Timelines

Hide Replies

Search Tweets

Filtered Streams

Sampled Streams

Twitter API v2 / Tweet Lookup / Single Tweet

Save

Send

GEThttps://api.twitter.com/2/tweets/id

ParamsAuthorizationHeadersBodyPre-request ScriptsTestsSettingsCookies

Query params

Key	Value	description
<input type="checkbox"/> tweet.fields		attachments,author_id,context_annotations,conversatio...
<input type="checkbox"/> expansions		Comma-separated list of fields to expand. Expansions e...
<input type="checkbox"/> media.fields		duration_ms,height,media_key,non_public_metrics,organ...
<input type="checkbox"/> poll.fields		Comma-separated list of fields for the poll object. Expl...
<input type="checkbox"/> place.fields		Comma-separated list of fields for the place object. Exp...
<input type="checkbox"/> user.fields		Comma-separated list of fields for the user object. Expl...

Path Variables

Key	Value	description
<input checked="" type="checkbox"/> id	1403216129661628420	Required. Enter a single Tweet ID.

BodyCookiesHeadersTest Results

200 OK468 ms734 BSave Response▼

PrettyRawPreviewVisualizeJSON▼

```
1 {
2   "data": {
3     "id": "1403216129661628420",
4     "text": "Donovan Mitchell went down after a collision with Paul George toward the
5   end of Game 2. https://t.co/Y9ihKhDLdN"
6   }
}
```

Documentation

https://api.twitter.com/2/tweets/id

This endpoint returns details about the Tweet specified by the requested ID.

For full details, see the [API reference](#) for this endpoint.

Authorisation Bearer token

This request is using an authorization helper from collection [Twitter API v2](#)

Request params

tweet.fields Comma-separated list of fields for the Tweet object.

Allowed values:
attachments,author_id,context_annotation
s,conversation_id,created_at,entities,geo,j
d,in_reply_to_user_id,lang,non_public_metr
ics,organic_metrics,possibly_sensitive,pro
moted_metrics,public_metrics,referenced_
tweets,reply_settings,source,text,withheld

Default values: id,text

OAuth1.0a User Context authorization
required if any of the following fields are
included in the request:
non_public_metrics,organic_metrics,promo

[View complete collection documentation →](#)

Find and ReplaceConsole

Desktop agentBootcampRunnerTrash

Postman. API Testing

<https://github.com/vdespa/introduction-to-postman-course/blob/main/simple-books-api.md>

☰ 128 lines (76 sloc) | 2.16 KB

Simple Books API

This API allows you to reserve a book.

The API is available at `https://simple-books-api.glitch.me`

Endpoints

Status

GET `/status`

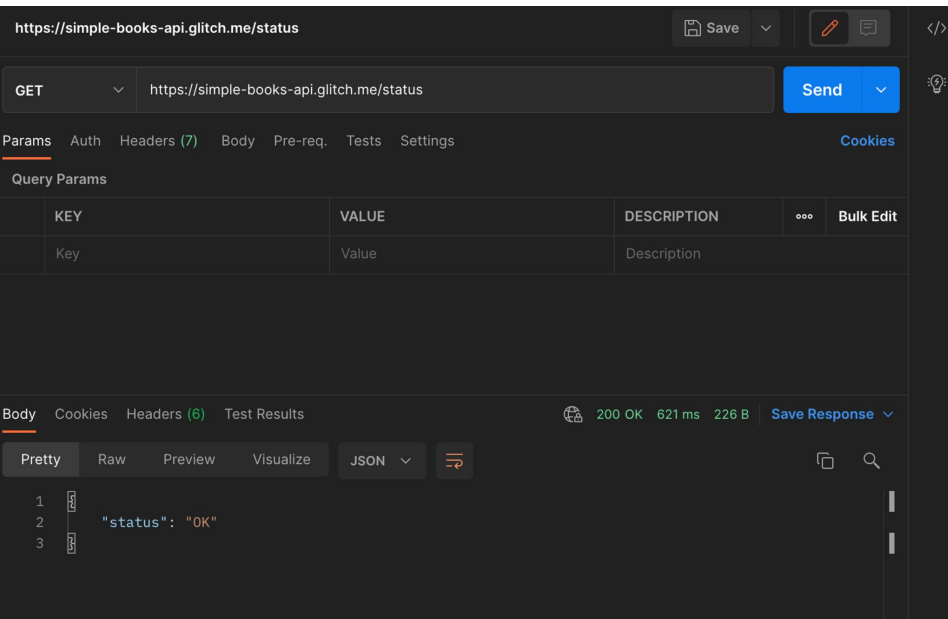
Returns the status of the API.

List of books

GET `/books`

Returns a list of books.

Postman. API Testing



https://simple-books-api.glitch.me/status

GET https://simple-books-api.glitch.me/status

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

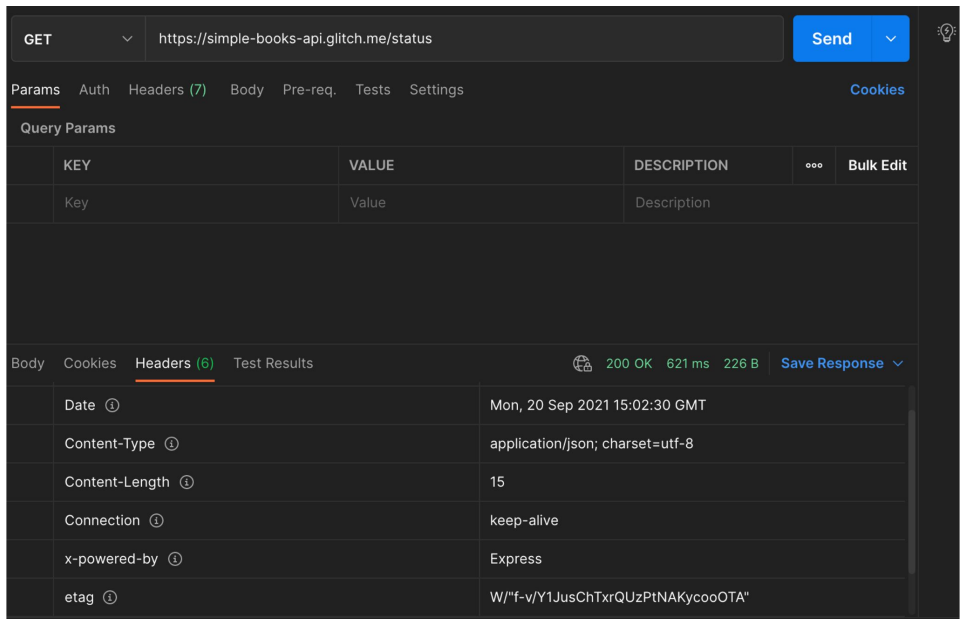
Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (6) Test Results 200 OK 621 ms 226 B Save Response

Pretty Raw Preview Visualize JSON

```
1 [25]  
2 "status": "OK"  
3 [25]
```



GET https://simple-books-api.glitch.me/status

Send

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

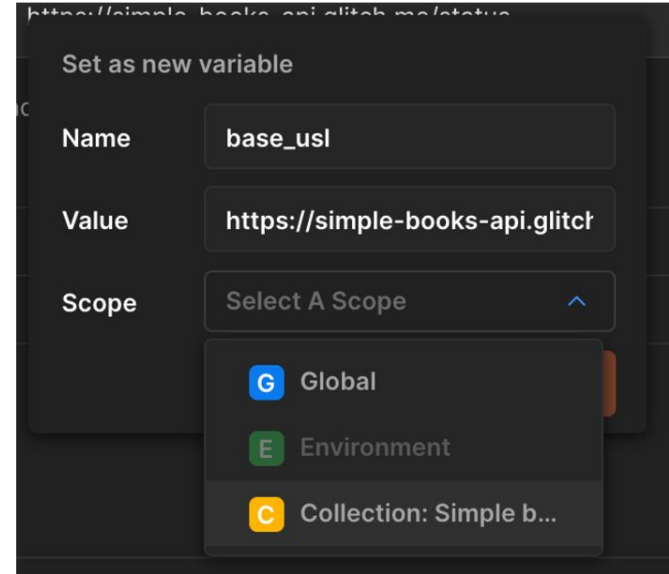
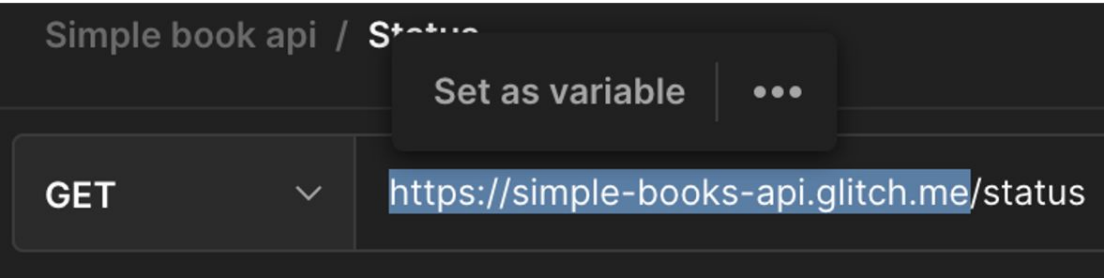
Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

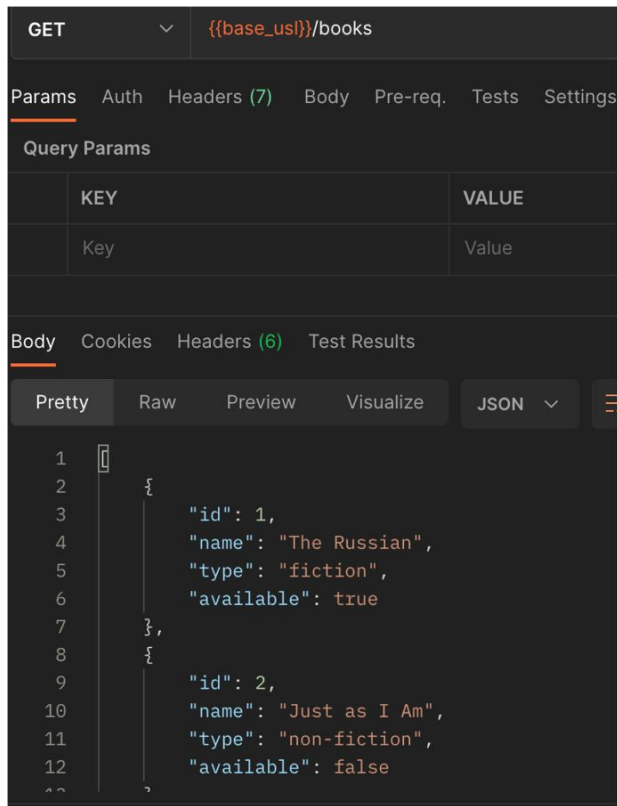
Body Cookies Headers (6) Test Results 200 OK 621 ms 226 B Save Response

Header	Value
Date	Mon, 20 Sep 2021 15:02:30 GMT
Content-Type	application/json; charset=utf-8
Content-Length	15
Connection	keep-alive
x-powered-by	Express
etag	W/"f-v/Y1JusChTxrQUzPtINAKycooOTA"

Postman. API Testing



Postman. API Testing



GET `{{base_url}}/books`

Params Auth Headers (7) Body Pre-req. Tests Settings

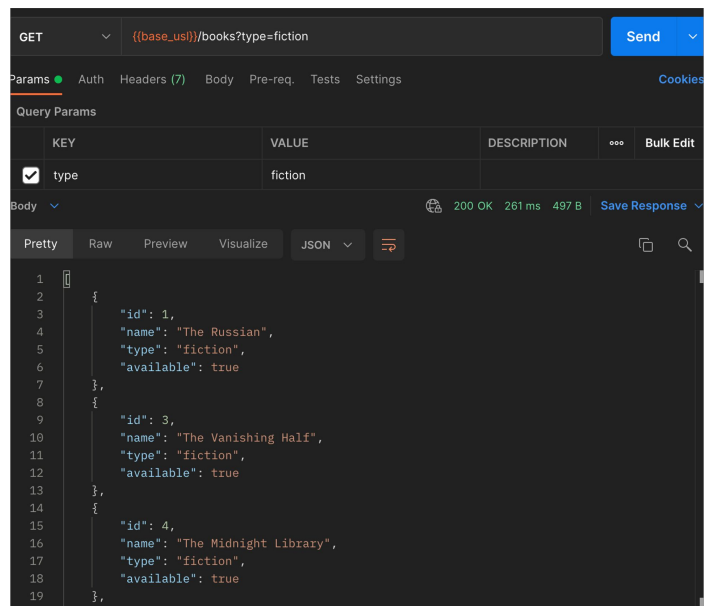
Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (6) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   {
3     "id": 1,
4     "name": "The Russian",
5     "type": "fiction",
6     "available": true
7   },
8   {
9     "id": 2,
10    "name": "Just as I Am",
11    "type": "non-fiction",
12    "available": false
13  }
14 }
```



GET `{{base_url}}/books?type=fiction` Send

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

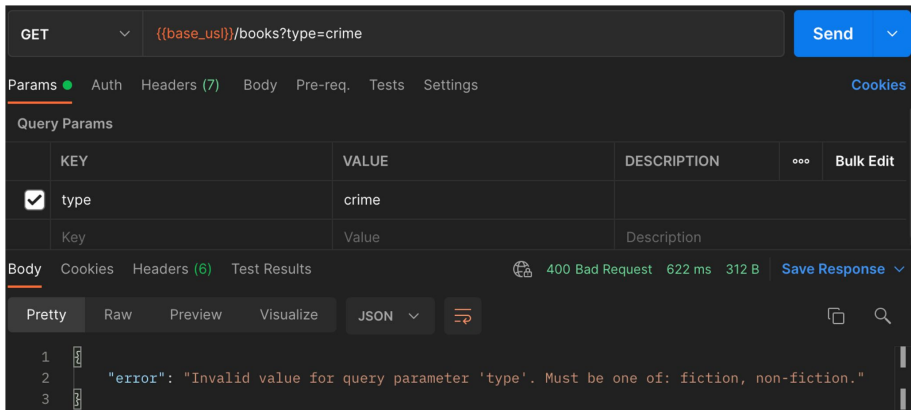
Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/> type	fiction			
Key	Value	Description		

Body 200 OK 261 ms 497 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   {
3     "id": 1,
4     "name": "The Russian",
5     "type": "fiction",
6     "available": true
7   },
8   {
9     "id": 3,
10    "name": "The Vanishing Half",
11    "type": "fiction",
12    "available": true
13  },
14  {
15    "id": 4,
16    "name": "The Midnight Library",
17    "type": "fiction",
18    "available": true
19  }
20 }
```



GET `{{base_url}}/books?type=crime` Send

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/> type	crime			
Key	Value	Description		

Body 400 Bad Request 622 ms 312 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "error": "Invalid value for query parameter 'type'. Must be one of: fiction, non-fiction."
3 }
```

Postman. API Testing

Submit an order

POST `/orders`

Allows you to submit a new order. Requires authentication.

The request body needs to be in JSON format and include the following properties:

- `bookId` - Integer - Required
- `customerName` - String - Required

API Authentication

To submit or view an order, you need to register your API client.

POST `/api-clients/`

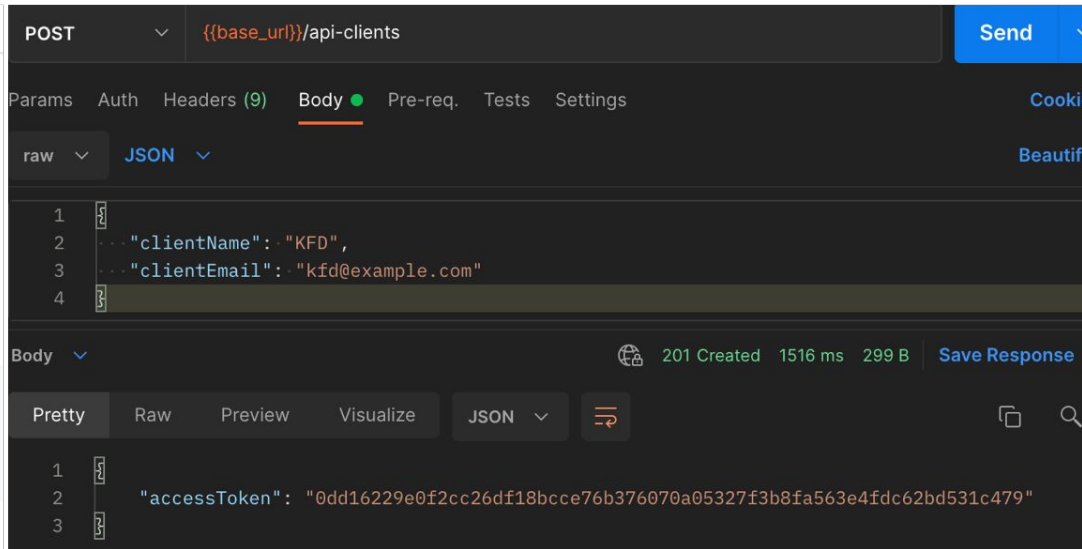
The request body needs to be in JSON format and include the following properties:

- `clientName` - String
- `clientEmail` - String

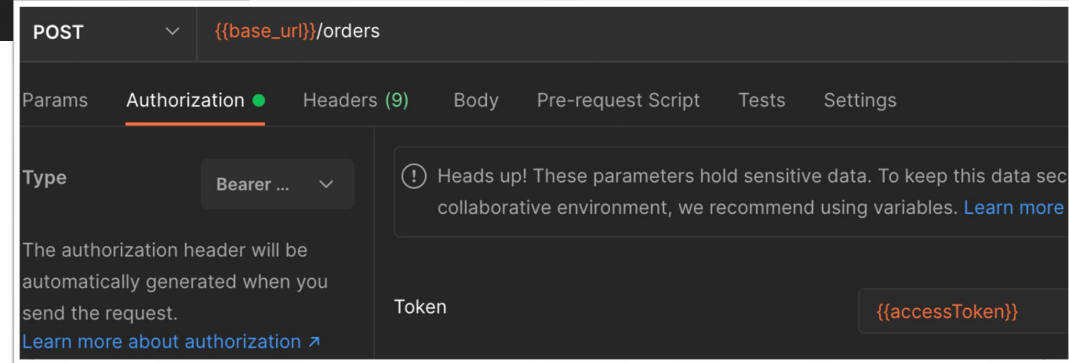
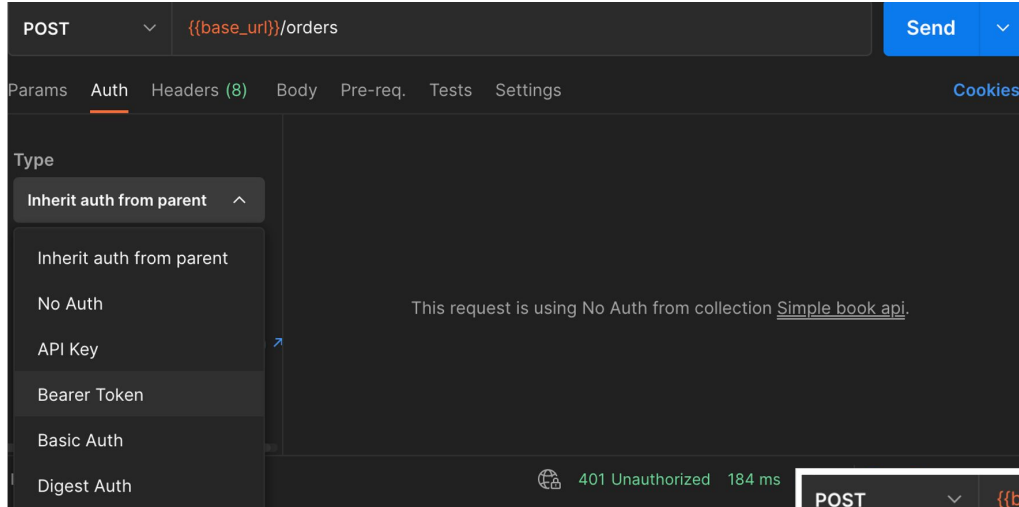
Example

```
{
  "clientName": "Valentin",
  "clientEmail": "valentin@example.com"
}
```

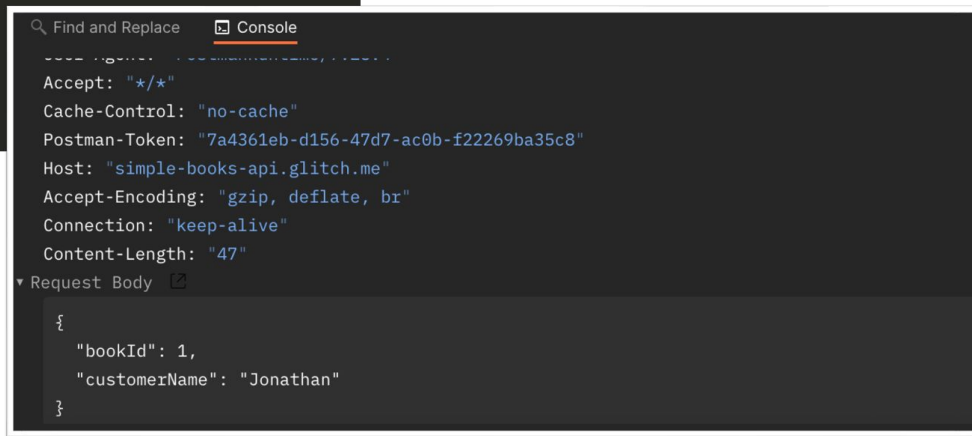
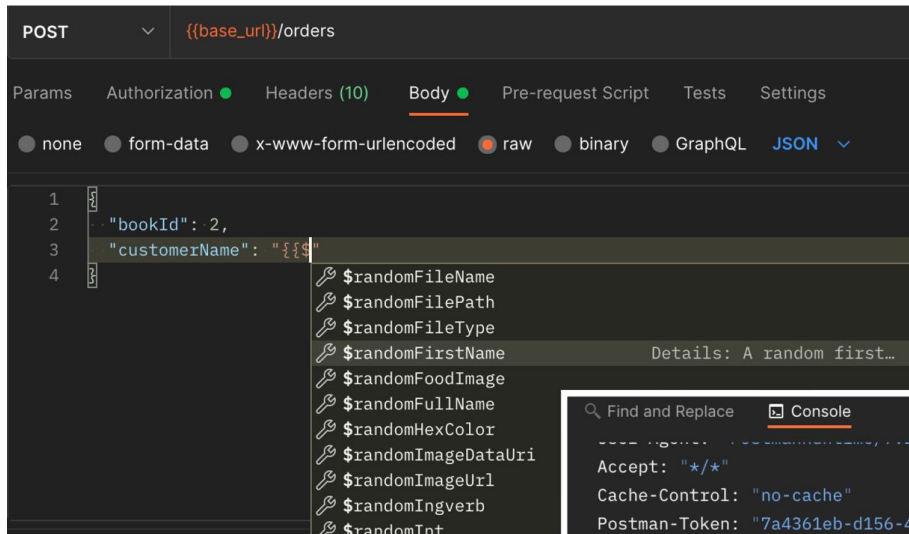
The response body will contain the access token.



Postman. API Testing



Postman. API Testing



Темы для докладов

1) vite vs webpack.

<https://medium.com/trendyol-tech/vite-webpack-killer-or-something-else-87019b4aec2>

2) npm vs yarn.

<https://www.knowledgehut.com/blog/web-development/yarn-vs-npm#which-one-to-choose?%C2%A0>

3) CORS