В этом уроке познакомимся поближе с темой модулей в JavaScript, научимся импортировать и экспортировать модули, а также рассмотрим пример простого модульного приложения.

Модули в JavaScript

По мере роста приложения и количества кода в нём, встает вопрос о разбиении приложения на отдельные части — модули. Обычно модуль представляет собой один файл, который содержит класс или библиотеку функций. С грамотно разбитой на модули программой удобнее работать в команде и её легче поддерживать.

По сути, модулем в JavaScript является любой файл с кодом. Импортировать другой файл можно с помощью инструкции `import`:

```
import './file.js';
```

При этом будет выполнен код из файла `file.js`, однако нельзя будет использовать элементы из этого файла. Чтобы использовать переменные, функции или классы из другого файла, нужно их сначала экспортировать с помощью инструкции `export`. Для примера, экспортируем простую функцию:

```
// module.js
export function sayHello() {
  alert("Hello");
}
```

Использовать эту функцию в другом файле можно, импортировав предварительно с помощью инструкции `import`:

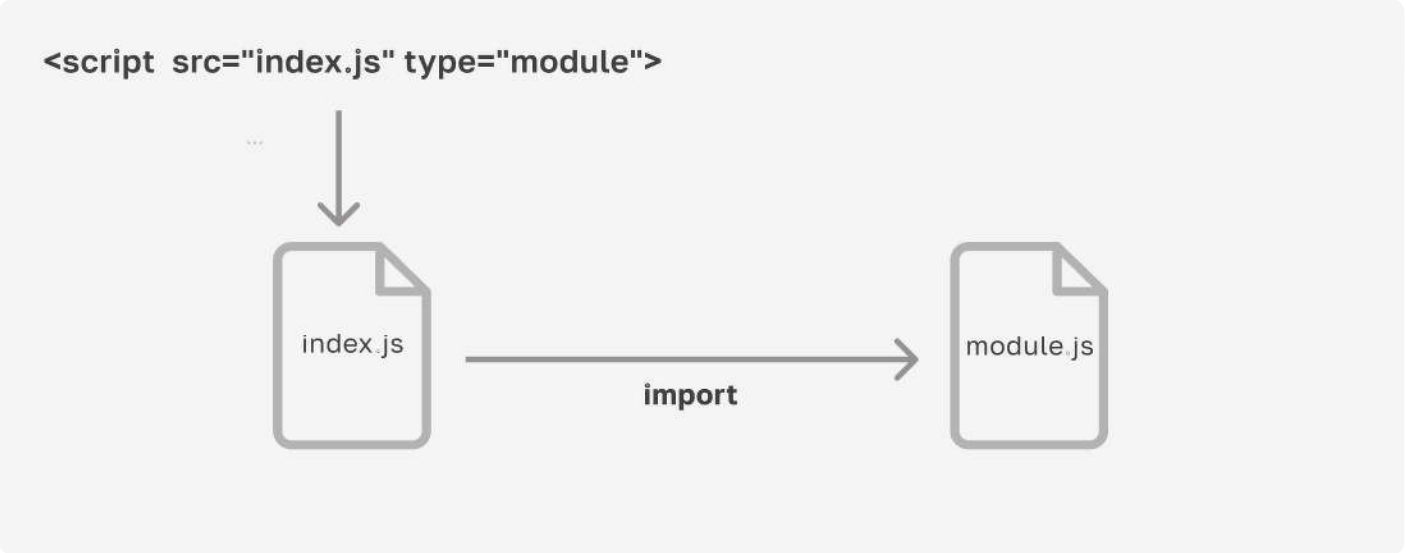
```
// index.js
import { sayHello } from "./module.js";
sayHello();
```

Мы разберём подробнее использование инструкций `export` и `import` далее.

К HTML-файлам ES-модули подключаются с помощью тегов `<script>` , как и обычные скрипты, добавляется только атрибут `type="module"` :

```
<script type="module" src="index.js"></script>
```

Так мы указали браузеру, что файл `index.js` должен быть проанализирован как модуль. Таким образом, внутри `index.js` и во всех дочерних файлах будет работать `import` / `export` (так как любой импорт также является модулем).



Такие модули имеют следующие особенности:

- Работают через протокол HTTP(S). Для работы с модулями нужен веб-сервер, поэтому для тестирования будем использовать локальный сервер.
- Внутри модулей всегда используется строгий режим (`'use strict'`).
- Переменные изолированы. Объявленные в модуле переменные и функции **не будут видны в других скриптах**. Чтобы поделиться чем-то, нужно использовать `import` / `export` .
- Загружаются отложено, они не будет блокировать отрисовку страницы. Их выполнение начнётся только после того, как документ загрузится полностью, аналогично использованию атрибута `defer` при подключении скрипта.
- Позволяют использовать `await` на верхнем уровне.

Однако на сегодняшний день ES-модули пока используются не так часто, но тебе всё равно следует о них знать. Сейчас чаще используются сборщики (по типу Webpack, Rollup). При их использовании часть этих особенностей отпадает, так как

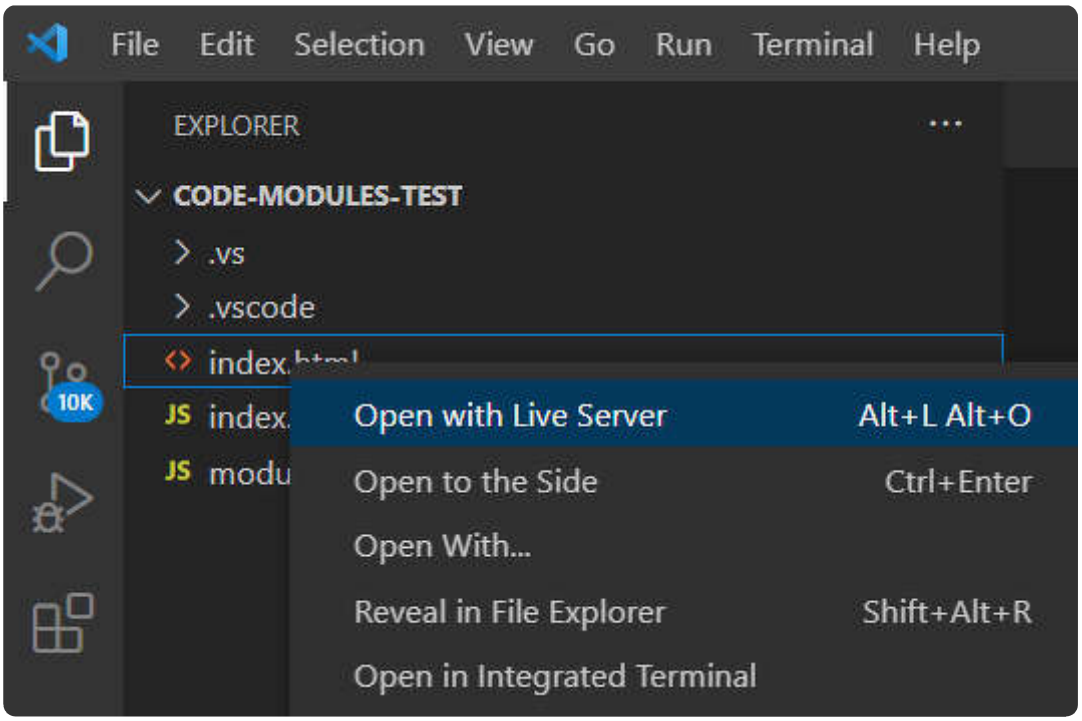
сборщик может объединять все модули в один файл. В частности, страницу с этим подключенным файлом можно открыть локально, без поднятия сервера.

Локальный веб-сервер

Для работы с `import` / `export` можно использовать один из сборщиков из предыдущего урока, либо использовать ES-модули и локальный веб-сервер. Давайте попрактикуемся на модулях.

Для простоты установим расширение **Live Server** для Visual Studio Code. Это можно сделать на странице расширения, либо из меню **View > Extensions** в VS Code.

После установки при нажатии правой кнопкой мыши на HTML-файлах в обозревателе файлов VS Code появится пункт “Open with Live Server”:



При выборе этого пункта будет запущен локальный сервер, и затем выбранная страница откроется в браузере.

Экспорт и импорт

Импортировать можно только те части кода, которые были помечены ключевым словом `export` .

Экспортировать можно что угодно — переменную, функцию или класс:

```
// Экспорт переменной
export const PI = 3.14;

// Экспорт функции
export function sayHello() {
  console.log("Hello");
}

// Экспорт класса
export class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }

  showInfo() {
    console.log(name + " " + year);
  }
}
```

Альтернативный способ экспорта заключается в перечислении экспортируемых элементов в виде списка внутри фигурных скобок после ключевого слова `export` :

```
export const PI = 3.14;
function sayHello() {}
class Car {}

export { PI, sayHello, Car };
```

Импортировать эти элементы можно с помощью ключевого слова `import` и списка нужных элементов в фигурных скобках. Затем следует ключевое слово `from` и путь к модулю, из которого делается импорт:

```
import { PI, sayHello, Car } from "./module.js";

console.log(PI);

sayHello();

const car = new Car("BWM X5", 2020);
car.showInfo();
```

Обратите внимание, что названия элементов должны быть такими же, как они объявлены в исходном модуле. Это так называемый "именованный импорт".

При импорте элементов может возникнуть конфликт имён с элементами текущего или других модулей. Для присвоения других названий импортируемым элементам есть ключевое слово `as` :

```
// Импортируем и устанавливаем название NumberPi
import { PI as NumberPi } from "./module.js";

// Переменная созданная в этом файле
const PI = 3.1415926;

// Переменная из module.js
console.log(NumberPi); // 3.14
```

Также можно импортировать все элементы из модуля с помощью символа `*` и слова `as` , после которого следует пользовательское название для модуля. Обращение к элементам делается через точку после названия модуля, например:

```
import * as MyModule from "./module.js";

console.log(MyModule.PI);

MyModule.sayHello();

const car = new MyModule.Car("BWM X5", 2020);
car.showInfo();
```

Экспорт по умолчанию

Ещё один способ экспорта — экспорт по умолчанию с помощью ключевых слов `export default` :

```
export default function sum(a, b) {
  return a + b;
}
```

Либо:

```
function sum(a, b) {
  return a + b;
}

export default sum;
```

Экспортировать таким способом можно только один элемент модуля. То есть модуль может содержать только одну инструкцию `export default` .

Для экспортированных таким способом элементов импорт делается без фигурных скобок:

```
import sum from "./module.js";
const result = sum(2, 3);
```

Для него также можно указать любое название при импорте:

```
import sumFunction from ".module.js";
const result = sumFunction(2, 3);
```

По сути, такой синтаксис импорта является сокращением от именованного варианта:

```
import { default as sum } from './module.js';
```

Экспорт по умолчанию можно также смешивать с обычным именованным экспортом:

```
export const PI = 3.14;

export default function sum(a, b) {
  return a + b;
}
```

Импорт:

```
import sum, { PI } from "./module.js";
```

Описание примера

Для демонстрации работы модулей доработаем пример, который мы использовали в уроке по методу `fetch()`. Там мы получали список пользователей через API и выводили его на экран:

Добавим возможность выбора пользователя и отображение детальной информации о нём. При этом используем модульный подход и также закрепим знания по темам ООП и `async` / `await`.

Структура проекта будет выглядеть так:

- Основной файл разметки `index.html`

- Стили — `index.css`
- Основной файл программы `src/index.js`
- Модули в папке `src/modules`

Разметка HTML

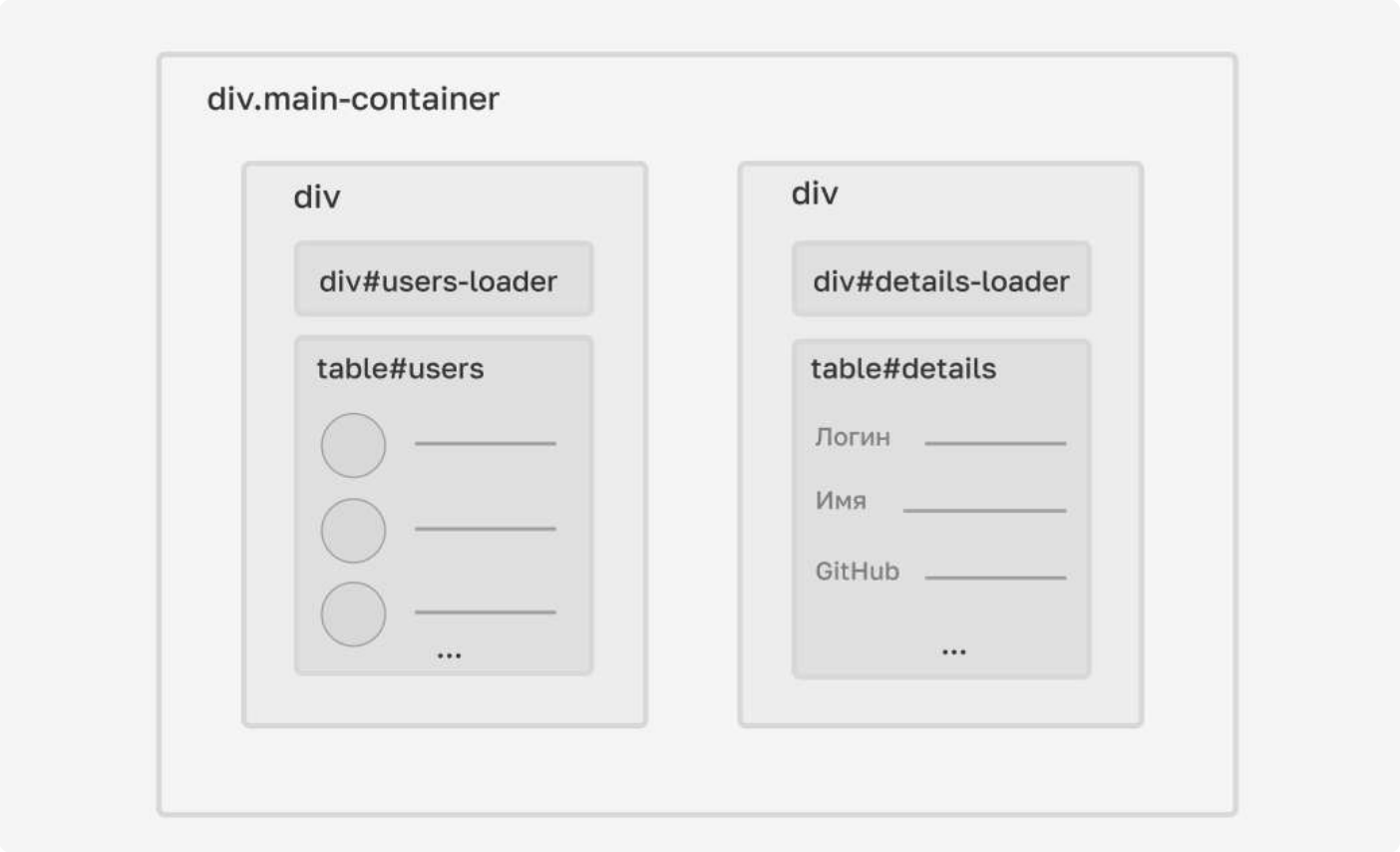
В качестве разметки используем следующий HTML:

```
<!DOCTYPE html>
<html>
<head>
  <title>Import/export example</title>
  <link rel="stylesheet" href="index.css">
</head>
<body>
  <div class="main-container">
    <div>
      <div id="users-loader" class="invisible">Загрузка...</div>
      <table id="users">
      </table>
    </div>
    <div>
      <div id="details-loader" class="invisible">Загрузка...</div>
      <table id="details">
      </table>
    </div>
  </div>
  <script type="module" src="./src/index.js"></script>
</body>
</html>
```

Он состоит из нескольких элементов:

- Блок `<div>` в качестве основного контейнера
- Два блока `<div>` для разделения экрана на две части — основной список и детали
- Блоки для отображения статуса загрузки (лоадеры)
- Таблица `#users` для отображения списка пользователей
- Таблица `#details` для отображения детальной информации о пользователе

Схематично разметка выглядит так:



Содержимое файла со стилями `index.css` :

```
.main-container {
  display: flex;
}

.invisible {
  display: none;
}

img {
  border-radius:50%;
  width: 50px;
}

#users tr:hover {
  opacity: 0.5;
  cursor: pointer;
}

#details {
  background-color: lightcyan;
}

#details td {
  padding: 5px;
}
```

Основной код

Рассмотрим код проекта сверху-вниз, от основных модулей к более низкоуровневым деталям. Хорошо написанный код должен читаться хорошо и без комментариев, однако мы всё равно будем разбирать основные моменты.

Содержимое основного файла с кодом `index.js` будет очень небольшим, так как основная функциональность будет подключена через модули:

```
import App from "./modules/app.js";

const app = new App();
app.run();
```

Здесь мы импортируем класс `App` из модуля `app.js`, затем создаем объект — экземпляр класса `App` и вызываем у него метод `run()`.

Содержимое модуля `app.js`:

```
import UsersBlock from "./users-block.js";

export default class App {
  #usersBlock;

  constructor() {
    this.#usersBlock = new UsersBlock();
  }

  run() {
    this.#usersBlock.showUsers();
  }
}
```

Разберём код этого модуля:

- В начале импортируется класс `UsersBlock` из модуля `users-block.js`.
- Затем объявляем класс `App`, который содержит:
  - Приватное свойство `#usersBlock` для блока со списком пользователей
  - Конструктор, в котором создается экземпляр класса `UsersBlock` и помещается в приватное свойство

- Метод `run()` , в котором вызывается метод отображения списка пользователей
- Класс помечен инструкцией `export default` , т.к. это единственный элемент для экспорта в модуле

Список пользователей

В модуле `users-block.js` запрашиваются данные о пользователях GitHub и формируется таблица со списком пользователей. Поскольку код представляет собой один класс, он экспортируется с помощью инструкции `export default` .

Код модуля:

```
import DetailsBlock from "./details-block.js";
import { createImg, insertRow, getRemoteData } from "./utils.js";

export default class UsersBlock {
  #table;
  #loader;

  constructor() {
    this.#table = document.querySelector("#users");
    this.#loader = document.querySelector("#users-loader");
    this.#loader.classList.toggle("invisible");
  }

  async showUsers() {
    try {
      const users = await getRemoteData("users?per_page=10");
      users.forEach((user) => {
        const elements = [createImg(user.avatar_url), user.login];
        insertRow(this.#table, elements, async () => {
          const details = new DetailsBlock(user.login);
          await details.showDetails();
        });
      });
    } catch (error) {
      console.error(error);
    } finally {
      this.#loader.classList.toggle("invisible");
    }
  }
}
```

Разбор модуля:

- Импортируем класс `DetailsBlock` и вспомогательные методы из `utils.js` .
- В конструкторе инициализируем приватные свойства:
  - `#table` — таблица, в которой мы будем выводить данные
  - `#loader` — элемент для отображения статуса загрузки (лоадер)
  - В методе `showUsers()` :
    - Получаем данные пользователей через API
    - Для каждого пользователя создаем отдельную строку в таблице с помощью вспомогательного метода `insertRow()` , который принимает в качестве параметров:
      - Таблицу для добавления строк.
      - Массив элементов для отображения в строке — в нашем случае это аватар пользователя и логин.
      - Коллбэк-функцию, которая будет вызываться при клике по строке. Здесь передается анонимная функция, в которой создается объект `DetailsBlock` с детальной информацией о пользователе, затем этот блок выводится на страницу.
    - Убираем надпись "Загрузка…" с экрана

Детальная информация о пользователе

В модуле `details-block.js` запрашиваются детальные данные о выбранном пользователе и формируется таблица с этими данными.

Структура кода модуля схожа с предыдущим модулем — инициализируем необходимые свойства в конструкторе, затем в единственном методе выводим информацию в таблицу.

Код модуля:

```
import { createLink, insertRow, getRemoteData } from "./utils.js";

export default class DetailsBlock {
  #userName;
  #table;
  #loader;

  constructor(userName) {
    this.#userName = userName;
```

```
    this.#table = document.querySelector("#details");
    this.#table.innerHTML = "";
    this.#loader = document.querySelector("#details-loader");
    this.#loader.classList.toggle("invisible");
  }

  async showDetails() {
    try {
      const details = await getRemoteData(`users/${this.#userName}`);

      const data = [
        ["Логин", details.login],
        ["Имя", details.name],
        ["GitHub", createLink(details.html_url)],
        ["Подписчики", details.followers],
        [
          "Дата регистрации",
          new Date(details.created_at).toLocaleDateString("ru-RU")
        ]
      ];

      data.forEach((d) => insertRow(this.#table, d));
    } catch (error) {
      console.error(error);
    } finally {
      this.#loader.classList.toggle("invisible");
    }
  }
}
```

Разбор модуля:

- Импортируем вспомогательные методы из `utils.js`
- В конструкторе инициализируем приватные свойства: имя пользователя (логин), таблицу для вывода данных и лоадер.
- В методе `showDetails()` запрашиваем данные с сервера, формируем массив с информацией для вывода на экран и для каждого элемента этого массива вставляем строки в таблицу.

Вспомогательные функции

В модуль `utils.js` вынесены вспомогательные функции. Эти функции удобно держать в отдельном файле, чтобы:

- не перегружать кодом логику основных классов — чем меньше деталей в коде, тем легче понять его смысл

- иметь возможность использовать эти функции в разных частях программы без необходимости копировать их

Код модуля состоит из четырёх функций:

```
export function createImg(src) {
  const img = new Image();
  img.src = src;
  return img;
}

export function createLink(url) {
  const link = document.createElement("a");
  link.textContent = url;
  link.href = url;
  link.target = "_blank";
  return link;
}

export function insertRow(table, elements, onclick) {
  const row = table.insertRow();
  for (const element of elements) {
    const column = row.insertCell();
    column.append(element);
  }
  row.onclick = onclick;
}

export async function getRemoteData(endPoint) {
  try {
    const apiUrl = "https://api.github.com/";
    const response = await fetch(apiUrl + endPoint);
    const result = await response.json();
    return result;
  }
  catch(ex) {
    console.log(ex);
  }
}
```

Здесь есть четыре функции:

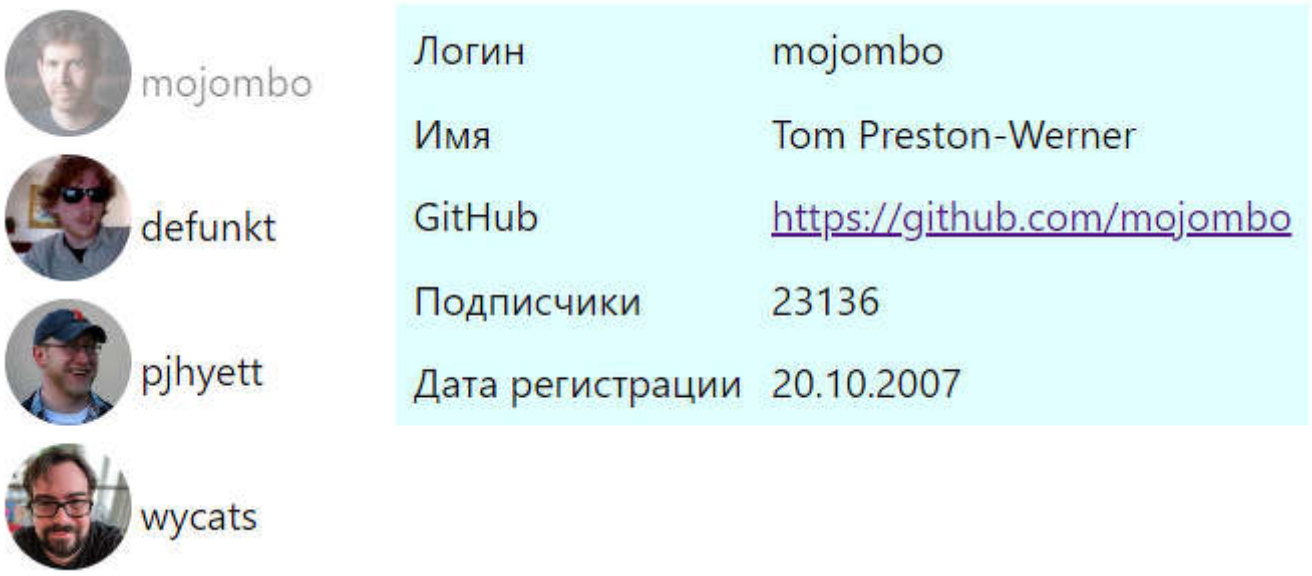
- `createImg(src)` — создает и возвращает картинку с указанным источником. Создание происходит при помощи конструктора `Image()`, что по функциональности эквивалентно методу `document.createElement('img')`.
- `createLink(url)` — создает и возвращает ссылку с указанным адресом, которая открывается в новом окне (`target="_blank"`)

- `insertRow(table, elements, onclick)` — вставляет в таблицу строку с указанными элементами (это может быть любой элемент HTML или просто строка). Для каждого элемента создается отдельный столбец. Для строки также может быть передан обработчик события `onclick`.
- `getRemoteData(endPoint)` — получает данные из API GitHub'a и парсит их в объект. Так как данные получаются по сети, функция сделана асинхронной — `async`. Соответственно, она возвращает объект `Promise` и вызывается в сочетании с ключевым словом `await` из других модулей.

Все функции помечены инструкцией `export`, чтобы иметь возможность импортировать их из других модулей.

Результат

В результате у нас получилась страница с возможностью выбрать пользователя из списка и посмотреть подробную информацию о нём (ссылка на CodeSandbox):



Поскольку страница сделана на основе модулей, её код легче читать, модифицировать и переиспользовать на других страницах.

Итоги

В уроке мы познакомились с модулями в JavaScript:

- Модуль — это как правило отдельный JS-файл, код из которого можно использовать в другом месте.
- Можно использовать в других местах только те элементы модуля, которые помечены ключевым словом `export`. При этом экспортировать можно любой элемент кода — переменную, функцию или класс.
- Есть несколько видов экспорта:
  - Перед объявлением элемента: `export const PI = 3.14;`
  - Отдельный экспорт:

    `export { PI }` или `export { PI as PiNumber }`
  - Экспорт по умолчанию:

    `export default const PI = 3.14`
- Импорт делается с помощью ключевого слова `import`. Существует несколько видов импорта.
  - Импорт именованных экспортов:

    `import { PI } from "module.js"`

    или `import { PI as PiNumber } from "module.js"`
  - Импорт по умолчанию:

    `import PI from "module.js"`
  - Импорт всех элементов модуля:

    `import * as MyModule from "module.js"`
- Сделанное на основе модулей приложение легче поддерживать, модифицировать и переиспользовать код из него.