Инициализация проекта

В этой части урока мы рассмотрим ещё один сборщик проектов под названием **Webpack**.

В качестве исходных файлов проекта можно использовать те же файлы, которые мы использовали для работы с Rollup — достаточно их скопировать в отдельную папку и работать как с новым проектом. Нам понадобится:

- `index.js`
- `index.html`
- `index.css`
- `assets/image.jpg` или любая другая картинка

После подготовки файлов инициализируем новый проект с помощью команды:

```
npm init -y
```

Установка Webpack

Порядок установки и работы с Webpack описан на официальном сайте сборщика.

Для начала установим два основных пакета с помощью команды:

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

Удостоверимся, что эти пакеты появились в `package.json` в секции `devDependencies` :

```
"devDependencies": {
  "webpack": "^5.74.0",
  "webpack-cli": "^4.10.0"
}
```

Далее требуется создать конфигурационный файл сборщика `webpack.config.js` со следующим содержимым:

```
const path = require('path');

module.exports = {
  mode: "development",
  entry: path.resolve(__dirname, 'index.js'),
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true
  }
};
```

В целом, конфиг похож на тот, который мы использовали в Rollup, но есть некоторые отличия:

- Пути до входного и выходного файлов мы прописали не строкой, а с помощью модуля path.
- Метод `path.resolve()` возвращает абсолютный путь к указанному файлу или папке. При этом переменная среды `__dirname` указывает на папку, в которой в данном случае находится `webpack.config.js` . Также в данном случае это корневая папка. Подробнее с этим мы поработаем, когда начнем изучать Node.js.

- `mode` показывает режим сборки – development или production. В режиме production файлы будут максимально оптимизированы и минифицированы, их будет сложно прочитать, поэтому пока поставим режим development.
- `entry` — основной файл с кодом, который служит в качестве входного файла для сборщика.
- `output.filename` и `output.path` — имя выходного файла и путь к нему.
- `output.clean` — указывает, нужно ли очистить директорию при сборке. В нашем случае, папка `dist` всегда будет очищаться перед каждой новой сборкой проекта. Таким образом, у нас там не будут скапливаться старые файлы.

HtmlWebpackPlugin

Плагин HtmlWebpackPlugin упрощает создание файлов HTML и может автоматически вставлять модули JavaScript в наш основной шаблон HTML.

Установим плагин с помощью команды:

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

Далее подключим плагин в конфиге `webpack.config.js` . Импортируем `HtmlWebpackPlugin` в начало конфига и добавим вызов конструктора `HtmlWebpackPlugin()` в секцию `plugins` :

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  // ...
  plugins: [
```

```
    new HtmlWebpackPlugin({
      template: path.resolve(__dirname, 'index.html'),
    })
  ]
};
```

Параметр `template` конструктора `HtmlWebpackPlugin()` указывает путь до нашего основного файла HTML.

После сборки можно будет сравнить исходный и выходной HTML-файлы. Для иллюстрации работы плагина, забежим вперёд и посмотрим на входной файл `index.html` :

```
<!DOCTYPE html>
<html>
<head>
  <title>Example page</title>
</head>
<body>
  <h1>Webpack test</h1>
</body>
</html>
```

И как бы выглядел выходной `dist/index.html` :

```
<!DOCTYPE html>
<html>
<head>
  <title>Example page</title>
  <script defer src="main.js"></script>
</head>
<body>
```

```
  <h1>Webpack test</h1>
</body>
</html>
```

Как видно, плагин автоматически подставил ссылку на собранный JS-файл. Далее по уроку мы научимся таким образом запускать сборку.

Обратите внимание, что собранные с помощью сборщика файлы HTML, JS и другие (лежат в папке `dist`) **не нужно редактировать**, так как они перезаписываются после сборки. Для внесения изменений в проект, мы редактируем **только** исходные файлы, затем запускаем сборку.

Babel

Для интеграции Babel в проект и, как следствие, получения поддержки старых браузеров установим babel-loader.

Установка производится командой:

```
npm install -D babel-loader @babel/core @babel/preset-env webpack
```

Помимо самого babel-loader, здесь мы также установили пакет ядра Babel (@babel/core) и пакет @babel/preset-env – умный пресет, который как раз и позволяет использовать последнюю версию JavaScript, подключая только нужные плагины, основываясь на браузерах, которые поддерживает конкретный проект.

В конфигурацию `webpack.config.js` в `module.exports` нужно вставить новую секцию `module`:

```
module.exports = {
  // ...
  // Другие настройки
  // ...
  module: {
    rules: [
      {
        test: /.m?js$/,
        exclude: /(node_modules|bower_components)/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        }
      }
    ]
  }
};
```

После сборки можно будет проверить. Благодаря этому плагину, например, такой код, соответствующий стандарту ES6 (ECMAScript 2015):

```
[1, 2, 3].map(n => n + 1);
```

будет преобразован в следующий в стандарте ES5 (ECMAScript 2009):

```
[1, 2, 3].map(function (n) {
  return n + 1;
```

```
});
```

Поддержка стилей

Чтобы Webpack мог включать в сборку файлы стилей, в соответствии с документацией нужно установить два пакета (style-loader и css-loader):

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

Далее добавим в `webpack.config.js` в `module.rules` новое правило сразу после правила, которое мы создали для Babel:

```
module: {
  rules: [
    // Существующее правило для babel-loader
    // ...
    {
      test: /.css$/i,
      use: ['style-loader', 'css-loader'],
    }
  ]
}
```

После этого нужно подключить CSS-файл в начало главного файла `index.js` :

```
import "./index.css";
```

В итоговой сборке стили будут подключены в блоке `<head>` основного HTML-файла:

```
<head>
  <title>Example page</title>
  <script defer="" src="main.js"></script>
  <style>
    body {
      background-color: black;
      color: white;
      padding: 0;
      margin: 0;
    }
    img {
      height: 300px;
      width: 300px;
    }
  </style>
</head>
```

Однако бывает удобнее иметь отдельные CSS-файлы со стилями в сборке. Для этого установим другой пакет — MiniCssExtractPlugin:

```
npm install --save-dev mini-css-extract-plugin
```

В `webpack.config.js` внесём три правки:

- В начале конфига добавим строку:

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
```

- В `plugins` добавим `new MiniCssExtractPlugin()` .
- В `module.rules` изменим правило для CSS. Заменим `style-loader` на `MiniCssExtractPlugin.loader` :

```
{
  test: /.css$/i,
  use: [MiniCssExtractPlugin.loader, "css-loader"]
}
```

Для удаления плагина `style-loader` из проекта используем команду:

```
npm uninstall style-loader
```

Теперь, после сборки, наши стили будет представлены в виде отдельного файла, подключенного в блоке `<head>` выходного HTML:

```
<link href="main.css" rel="stylesheet">
```

Поддержка картинок

Для поддержки картинок при сборке нужно добавить ещё одно правило в `webpack.config.js` :

```
module: {
  rules: [
    // Существующие правила
    // ...
    {
      test: /.(png|svg|jpg|jpeg|gif)$/i,
      type: 'asset/resource',
    }
  ]
}
```

Далее добавим картинку в `index.js` аналогичным образом, как мы делали это в части урока про Rollup, после она добавится в папку `dist` :

```
import MY_IMAGE from './assets/image.png';

// Остальной код

const img = document.createElement("img");
img.src = MY_IMAGE;
document.body.append(img);
```

Сборка проекта

Итак, сборщик Webpack настроен, и все нужные плагины установлены, можно попробовать запустить сборку.

Для этого в файле `package.json` добавим новую команду в секции `scripts` (остальные части файла пропущены):

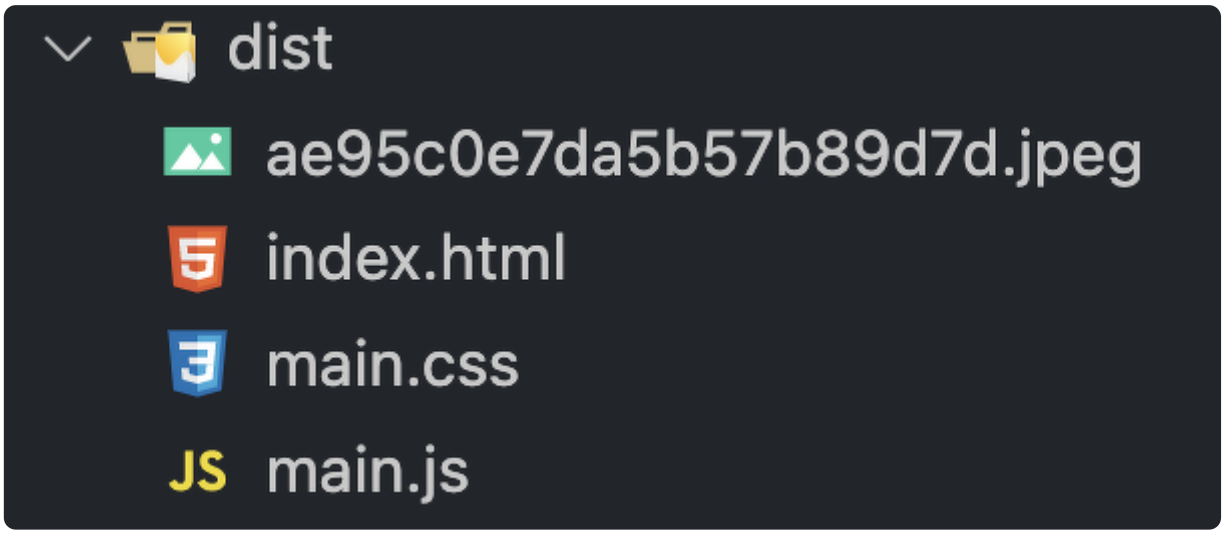
```
"scripts": {
  "start": "webpack"
}
```

Таким образом, мы создали скрипт с именем `start`, при запуске которого будет выполнена команда `webpack`.

Выполним наш скрипт `start` с помощью следующей команды в терминале:

```
npm run start
```

Если сборка прошла успешно, в проекте появится папка `dist` с подобным набором файлов:



Откроем файл `index.html` из папки `dist` в браузере и убедимся, что страница отображается корректно. Ещё раз напомним, что менять файлы в папке со сборкой (`dist`) не нужно. Для внесения изменений используем исходные файлы проекта, после чего запускаем сборку ещё раз.

Локальный сервер

В качестве финального штриха установим плагин, который позволит открывать нашу страницу на локальном веб-сервере и обновлять её при любых изменениях в проекте.

Плагин называется webpack-dev-server и устанавливается командой:

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

Далее, в соответствии с документацией, добавим новый блок настроек в конфигурационный файл `webpack.config.js` :

```
module.exports = {
  // ...
  // Другие настройки
  // ...
  devServer: {
    static: path.resolve(__dirname, 'dist'),
    port: 8080,
    open: true
  }
};
```

Настройки `devServer` :

- `static` – путь к сборке (в нашем случае, это папка `dist`)
- `port` – порт, на котором будет запускаться локальный сервер
- `open` – флаг, будет ли открываться странице при сборке

Обновим скрипты в package.json для запуска сборки:

```
"scripts": {
  "start": "webpack serve --mode=development",
  "build": "webpack --mode=production"
}
```

Описание данных скриптов:

- `start` — собирает проект и запускает локальный сервер в режиме для разработки.
  Также обратите внимание, что при запуске `start` , файлы будут храниться не в папке `dist` , а в оперативной памяти компьютера для наилучшей производительности. Теперь папка `dist` будет использоваться только для production-сборки.
- `build` — сборка в режиме production, в которой код минифицирован и оптимизирован. Отличается от сборки в режиме разработки, в которой удобнее работать с выходными JS-файлами, читать и отлаживать их.
  Но на текущий момент у нас не минифицируется CSS-код. Для этого, добавим CssMinimizerWebpackPlugin:

  ```
  npm install css-minimizer-webpack-plugin --save-dev
  ```

И в конфиге добавим:

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");

module.exports = {
  // Удалим mode (теперь задаем в скрпмах package.json)
  entry: path.resolve(__dirname, 'index.js'),
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: path.resolve(__dirname, 'index.html'),
    }),
    new MiniCssExtractPlugin()
  ],
  devServer: {
    static: path.resolve(__dirname, 'dist'),
    port: 8080,
    open: true
```

```
  },
  module: {
    rules: [
      {
        test: /.m?js$/,
        exclude: /(node_modules|bower_components)/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        }
      },
      {
        test: /.css$/i,
        use: [MiniCssExtractPlugin.loader, 'css-loader'],
      },
      {
        test: /.(png|svg|jpg|jpeg|gif)$/i,
        type: 'asset/resource',
      }
    ]
  },
  optimization: {
    minimize: true,
    minimizer: [new CssMinimizerPlugin(), '...'],
```

```
  },
};
```

Теперь при запуске команды `npm run start` (или также возможно `npm start`) через терминал будет происходить не только сборка проекта, но и запуск локального веб-сервера, а также открытие нашей страницы в браузере с поддержкой быстрого обновления при изменениях.

Запустим команду `npm run start` и убедимся, что изменения в исходных файлах проекта отображаются в браузере, значит, сборщик и все плагины настроены правильно.

При запуске `npm run build` будет создаваться базовая production-сборка проекта, при которой файлы минимизируются. Оптимизации для production – довольно объемная тема, при желании вы можете самостоятельно с ней ознакомиться здесь более подробно.

Итоги

Краткие итоги урока:

- **Сборщик проектов** — это инструмент, который собирает исходный код проекта в один или несколько файлов, оптимизирует, а также собирает и подключает другие ресурсы, такие как стили и картинки.
- Для работы со сборщиками необходим **Node.js** — среда выполнения JavaScript-кода вне браузера. Сборщики и плагины к ним подключаются с помощью менеджера пакетов **npm**, входящего в состав Node.js.
- **Rollup** — современный сборщик проектов с большим количеством плагинов, включая плагины для поддержки старых браузеров, работы со стилями и картинками, запуска локального сервера и т.д. Хорошо подходит для

разработки небольших приложений и библиотек.

- **Webpack** — самый популярный сборщик, имеющий богатый набор плагинов и очень гибкие настройки. Зарекомендовал себя как хороший сборщик для средних и больших приложений, имеющих большое количество зависимостей от сторонних библиотек.