

Занятие 1

План занятия

1. ПО для разработки
2. Семантическое Версионирование
3. nodeJS, npm, prx
4. Git
5. package.json
6. Webpack
7. Create React App
8. Readme.md

1. ПО для разработки

IDE и зачем она вам Основные функции IDE, которые пригодятся:

- компилятор: превращает ваш код в исполняемый файл;
- интерпретатор: запускает скрипты, которые не нужно компилировать;
- отладчик: позволяет находить проблемные места и ошибки в коде;
- инструменты автоматизации: помогают автоматизировать сборку проекта и ускорить процесс разработки.

Visual Studio Code (<https://code.visualstudio.com/download>)- Текстовый редактор, разработанный Microsoft для Windows, Linux и macOS. Позиционируется как «лёгкий» редактор кода для кроссплатформенной разработки веб- и облачных приложений. Включает в себя отладчик, инструменты для работы с Git, подсветку синтаксиса, IntelliSense и средства для рефакторинга.

WebStorm (<https://www.jetbrains.com/webstorm>)- Интегрированная среда разработки на JavaScript, CSS & HTML от компании JetBrains, разработанная на основе платформы IntelliJ IDEA. WebStorm обеспечивает автодополнение, анализ кода на лету, навигацию по коду, рефакторинг, отладку, и интеграцию с системами управления версиями.

Vim (<https://www.vim.org>)- Свободный текстовый редактор, созданный на основе более старого vi, разработанного Биллом Джойем. Автор Vim'а, Брам Моленар, создал его из порта редактора

Stevie для Amiga и в 1991 году выпустил общедоступную версию. Vim предназначен для использования как в интерфейсе командной строки, так и в качестве отдельного приложения в графическом пользовательском интерфейсе.

Plugins (расширения) для VS Code

(<https://code.visualstudio.com/docs/editor/extension-marketplace>)

- Live Server
- Git Lens
- Color Highlight
- Prettier
- Highlight Matching Tag
- Auto Rename Tag
- ESLint
- Auto Close Tag
- Bracket Pair Colorizer
- VS Code Icons
- Better Comments

2. Семантическое Версионирование

(<https://semver.org/lang/ru/>) Учитывая номер версии **МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ**, следует увеличивать:

МАЖОРНУЮ версию, когда сделаны обратно несовместимые изменения API. **МИНОРНУЮ** версию, когда вы добавляете новую функциональность, не нарушая обратной совместимости. **ПАТЧ-версию**, когда вы делаете обратно совместимые исправления. Дополнительные обозначения для предрелизных и билд-метаданных возможны как дополнения к **МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ** формату.

3. nodeJS, npm, npx

nodeJs (<https://nodejs.org/en>)- Программная платформа, основанная на движке V8, превращающая JavaScript из узкоспециализированного языка в язык общего назначения. Node.js добавляет возможность JavaScript взаимодействовать с устройствами ввода-вывода

через свой API, написанный на C++, подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода. Node.js применяется преимущественно на сервере, выполняя роль веб-сервера, но есть возможность разрабатывать на Node.js и десктопные оконные приложения и даже программировать микроконтроллеры. В основе Node.js лежит событийно-ориентированное и асинхронное программирование с неблокирующим вводом/выводом.

Для разработки необходимо выбирать **LTS** версии nodeJS. Для исследований новых функции nodeJS можно использовать версии **Current**

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

- CLI (интерфейс командной строки) – средство для размещения и скачивания пакетов,
- онлайн-репозитории (<https://www.npmjs.com>), содержащие JS пакеты. Сам по себе npm не запускает никаких пакетов. Если вы хотите запустить пакет, используя npm, вы должны указать этот пакет в своем файле package.json.

Когда исполняемые файлы устанавливаются через npm, он создает ссылки на них:

- локальные установки имеют ссылки, созданные в каталоге ./node_modules/.bin/
- глобальные установки имеют ссылки, созданные из глобального каталога bin/ (например: /usr/local/bin в Linux или в %AppData%/npm Windows) Выполнить пакет с помощью npm, вы должны либо ввести локальный путь, например: npm v create-react-app запустить локально установленный пакет, добавив его в свой файл package.json в разделе скриптов, например:

```
{
  "name": "your-application",
  "version": "1.0.0",
  "scripts": {
    "your-package": "your-package"
  }
}
```

Затем вы можете запустить скрипт, используя npm run: npm run your-package

nrpx - является инструментом CLI, цель которого - упростить установку и управление зависимостями, размещенными в реестре npm. Теперь очень легко запустить любой исполняемый файл Node.js, который вы обычно устанавливаете через npm. Вы можете

запустить следующую команду, чтобы увидеть, установлена ли она уже для вашей текущей версии npm: `which prx` Запустить локально установленный пакет легко: `prx your-package prx` проверит, существует ли `command` или `package` в `$PATH` локальных бинарных файлах проекта, или, если да, то выполнит его. **Выполнить пакеты, которые ранее не были установлены** важным преимуществом является возможность выполнения пакета, который ранее не был установлен. использовать некоторые инструменты CLI, но не хотите устанавливать их глобально, просто чтобы протестировать их. то означает, что вы можете сэкономить место на диске и просто запускать их только тогда, когда они вам нужны. Это также означает, что ваши глобальные переменные будут менее загрязнены.

4. Git

(<https://git-scm.com/downloads>) - распределённая система контроля версиями Подробнее про Git в бумаре - <https://www.chitai-gorod.ru/product/git-dlya-professionalnogo-programmista-podrobnое-opisanie-samoy-populyarnoy-sistemy-kontrolya-versiy-2483809?productShelf=&shelfIndex=0&productIndex=0> **Создание репозитория** `git init` - Создаёт git репозитории и директорию `.git` в текущей директории `git clone` - Клонировать репозитории с названием **Состояние файлов** `git status` - на какой ветке вы сейчас находитесь и состояние всех файлов **Работа с индексом** `git add` - добавить файлы в индекс `git restore` - удалить из индекса некоторые файлы **Работа с коммитами** `git commit` - сделать коммит ваших изменений **Просмотр истории** `git log` - просматривать историю коммитов вашего репозитория **Работа с удалённым репозиторием** `git remote add` - связь с внешними репозиториями используйте `git push` - отправки данных на сервер

Выжимка по основам работы с git - <https://habr.com/ru/articles/472600/>

5. package.json

(<https://docs.npmjs.com/cli/v9/configuring-npm/package-json>) это файл в формате JSON, который содержит информацию о вашем проекте. **Важен по нескольким причинам:**

- Служит централизованным местом для управления всеми зависимостями вашего проекта. Это позволяет легко отслеживать, какие пакеты вам нужны и какие версии вы используете.
- Позволяет автоматизировать такие задачи, как создание и тестирование кода, путем определения скриптов в разделе скриптов файла.
- Дает другим возможность понять, что представляет собой ваш проект, что он делает и какой вклад они могут внести. **Создание файла package.json** `npm init` - команда задаст вам ряд вопросов о вашем проекте и сгенерирует для вас базовый файл `package.json`

Управление зависимостями В разделах Dependencies - перечислены пакеты, которые необходимо запустить вашему проекту `npm install` - установка пакета `npm update` - обновление пакета **Скрипты в package.json** Сценарии, которые можно запускать с помощью команды `npm run`. чтобы определить скрипт, просто добавьте пару ключ-значение в раздел скриптов вашего файла `package.json`.

```
"scripts": {  
  "test": "jest"  
}
```

запустить тестовый скрипт, выполнив следующую команду - `npm run test` **Основные поля в package.json** Содержит несколько важных полей, в том числе:

- `name`: Название вашего проекта. Это имя должно быть уникальным в реестре NPM.
- `version`: номер версии вашего проекта в формате `x.y.z`.
- `description`: краткое описание вашего проекта.
- `dependencies`: список зависимостей, необходимых для запуска вашего проекта. Эти зависимости устанавливаются NPM и перечислены в разделе зависимостей вашего файла `package.json`.
- `scripts`: список скриптов, которые можно запустить с помощью команды `npm run`.
- `devDependencies`: список зависимостей, которые необходимы только для разработки, например инструментов тестирования или сборки. **Пример package.json для проекта на React**

```
{  
  "name": "my-react-app",  
  "version": "1.0.0",  
  "description": "A simple React application",  
  "dependencies": {  
    "react": "16.13.1",  
    "react-dom": "16.13.1"  
  },  
  "devDependencies": {  
    "babel-core": "6.26.3",  
    "babel-loader": "7.1.5",  
    "babel-preset-env": "1.7.0",  
    "babel-preset-react": "6.24.1",  
    "webpack": "4.43.0",  
    "webpack-dev-server": "3.11.0"  
  }  
}
```

```
f,
"scripts": {
  "start": "webpack-dev-server --mode development --open",
  "build": "webpack --mode production"
}
}
```

6. webpack

это статический сборщик модулей JavaScript с открытым исходным кодом. Он создан в первую очередь для JavaScript, но может преобразовывать внешние ресурсы, такие как HTML, CSS и изображения, если включены соответствующие загрузчики. webpack принимает модули с зависимостями и генерирует статические ресурсы, представляющие эти модули.

(<https://webpack.js.org/guides/getting-started/>)

Какую проблему решает Webpack решает проблему вечного подключения библиотек и фреймворков к HTML. В каком порядке их подключать, как решать конфликты, как сразу же оптимизировать картинки, как включать css в JavaScript, подобно тому, как это делается в React? Всеми этими вопросами занимается Webpack

Основные термины

- Конфигурационный файл (Configuration)
- Входная точка (Entry)
- Точка выхода (Output)
- Загрузчики (Loaders)
- Плагины (Plugins)
- Режимы (Modes)

Конфигурационный файл - webpack.config.js

```
module.exports = {
  // Конфигурация
};
```

Для PROD создают отдельный файл webpack.prod.js. Для DEV создают отдельный файл webpack.dev.js. В package.json нужно просто указать команды для сборки, чтобы Webpack знал какую конфигурацию ему таскать:

```

{
  ../
  "scripts": {
    "build:dev": "webpack --config webpack.dev.js",
    "build:prod": "webpack --config webpack.prod.js"
  }
  ../
}

```

Входная точка - Webpack строит граф (дерево, в случае если входной файл - один) зависимостей. Он проходится по всем импортам внутри ваших файлов, строит граф, а затем начинает импортировать все прямо в финальный файл. Файл, с которого Webpack начнет построение этого графа. Это файл, который нам нужно включить первым и от которого будут идти все импорты. Обычно данный файл является `./src/index.js`, однако данную входную точку конечно же можно поменять:

```

module.exports = {
  entry: './path/to/index.js',
};

```

Точка выхода - Webpack отдает нам один файл. По умолчанию этот файл находится в `./dist/main.js`, однако вы можете также поменять это с помощью конфигурации:

```

// Импортируем модуль для работ с путями
const path = require('path');

module.exports = {

  // Указываем входную точку
  entry: './path/to/index.js',

  // Указываем точку выхода
  output: {

    // Тут мы указываем полный путь к директории, где будет храниться конечный
    файл
  }
}

```

```
    path: path.resolve(__dirname, 'dist'),

    // Указываем имя этого файла
    filename: 'my-first-webpack.bundle.js',
  },
};
```

Webpack может упаковывать все что угодно, даже картинки формата png. Сами картинки он как не странно в код не запихнет, однако он перенесет их. Для переноса ему нужен точный путь к директории, где будет точка выхода, чтобы построить правильный путь. Имя файла и путь мы разделяем именно поэтому. **Loaders (Ладеры)** Из коробки Webpack понимает только JavaScript и JSON, что будет если мы попытаемся запихнуть внутрь cjs, ejs, html, typescript? Ничего, Webpack отдаст нам ошибку. Для таких вещей придумали ладеры. **Ладеры** — специальные модули, которые созданы для того, чтобы считывать и обрабатывать файлы. Для ладеров в свою очередь придумали правила. **Правила** — это в свою очередь просто объекты в конфигурации, которые указывают на файлы, которые нужно обработать и указывают ладер. Все ладеры указываются в конфигурации следующим образом:

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

module.exports = {
  entry: './path/to/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),

    filename: 'my-first-webpack.bundle.js',
  },

  // Указываем тут, что будем использовать спец. модуль для определенных файлов
  (ладер)
  module: {

    // Указываем правила для данных модулей
    rules: [

      // Указываем правило для каждого ладера
      {test: /\.txt$/, use: 'raw-loader'},
    ],
  },
};
```


Как можно увидеть есть свойство `module`, которое содержит в себе правила, массив `rules` в свою очередь уже содержит сами правила: в него включаются правила в виде объекта, где `test` - путь в виде Regex, `use` - сам ладер, который должен быть использован. Стоит уточнить, что перед тем как использовать ладеры — их надо найти и скачать с помощью NPM `npm i -D raw-loader` Существует очень много ладеров, от CSS и Sass до png и svg. Ладеры в свою очередь могут таскать больше чем просто файлы `.txt`, естественно. Они используются для той же компиляции TypeScript:

```
module.exports = {
  module: {
    rules: [
      { test: /\.ts$/, use: 'ts-loader' },
    ],
  },
};
```

перед тем как использовать ладер - его нужно установить - `npm i -D ts-loader` **Процессинг файлов с помощью ладеров** Допустим у нас есть задачка: мы хотим процессить CSS-файлы с помощью SASS, а также включать их в JavaScript. Реализация у данной идеи будет следующая:

```
module.exports = {
  module: {
    rules: [

      // Правило для CSS
      {
        test: /\.css$/,
        use: [
          {loader: 'style-loader'},
          {loader: 'css-loader'},
          {loader: 'sass-loader'}
        ]
      }
    ]
  }
}
```

можем указывать несколько лоадеров для одного типа файла, помещая все лоадеры в массив. Все лоадеры будут выполняться последовательно. **Для плагинов последовательности и сам процессинг работает точно также**

Plugins (Плагины) Существует очень много лоадеров, от CSS и Sass до png и svg. Плагины — внешние модули для Webpack, которые позволяют управлять и обрабатывать файлы, которые не импортируются в JavaScript. Пример, где мы будем импортировать index.html, это позволит не держать его в директории dist (то есть саму директорию можно будет удалять и ничего не потеряем), а также вставлять внутрь переменные окружения и пути к файлам. Конфигурация (для упрощения часть кода с входной и выходной точкой - пропущена):

```
// Для того чтобы достучаться до плагина
const HtmlWebpackPlugin = require('html-webpack-plugin');

// У самого Webpack уже есть встроенные плагины, их неплохо тоже импортировать
const webpack = require('webpack');

module.exports = {
  module: {
    rules: [{ test: /\.txt$/, use: 'raw-loader' }],
  },

  // Указываем новые плагины для обработки файлов
  plugins: [

    // Указываем что будем обрабатывать HTML с помощью плагина
    new HtmlWebpackPlugin({ template: './src/index.html' })
  ],
};
```

используем плагин html-webpack-plugin для того чтобы обрабатывать HTML. В данном случае он просто перенесет файл из ./src в ./dist **Mode (Режим)** По умолчанию без конфигурации у Webpack нет режима — none. Если мы напишем конфигурацию, то Webpack настоятельно посоветует

```
module.exports = {
  mode: 'production',
};
```

У Webpack существуют два режима:

- Режим для разработки — development. Данный режим как не сложно догадаться рассчитан на разработку, он не сильно сжимает бандл (точку выхода, файл который получится в конце), а также может привязывать карты исходников и кучу других мелких фиш
- Режим продакшена — production. Режим, который будет сжимать ваш бандл, покуда сможет и делать совершенно нечитабельный (такой, что и обфускатор не потребуется), но оптимизированный код. **Две и более входные точки** можем разделять файлы на две и более входных точки. Мы можем компилировать их в отдельные точки выхода, а также в один. С помощью данной конфигурации можно указать входные файлы:

```
module.exports = {
  entry: {
    main: './src/main.js',
    vendor: './src/vendor.js',
  },
};
```

можем передать в entry не объект с названиями файлов, а массив:

```
module.exports = {
  entry: ['./src/main.js', './src/vendor.js'],
};
```

Собирать данные исходники:

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

module.exports = {
  output: {
    path: path.resolve(__dirname, 'dist'),

    // Тут мы указываем, что будем компилировать каждый файл с таким же именем, но
    с постфиксом bundle.js
    filename: '[name].bundle.js',
  },
};
```

[name] — в данном случае имя входного файла Данная строка является шаблонной (не в привычном понимании JavaScript), ибо внутри квадратных скобок мы указываем шаблон наименования. Все шаблоны можно посмотреть -

<https://webpack.js.org/configuration/output/#template-strings> Одним из полезных свойств является [contenthash]. Он используется в основном для продакшена, чтобы каждый раз при компиляции создавался новый хэш (хэш создается по контенту в файле, так что если файл изменился, то хэш будет новый). Данное свойство активно используется для обхода кэширования файлов в браузерах. Дело в том, что браузер не будет грузить все ваши файлы с одним и тем же названием каждый раз, как вы переходите на сайт, ибо это очень долго и влияет на время загрузки, даже если файлы поменялись. Проблема решается "в лоб", просто меняем название файла, если он сам изменился. **Один выходной файл** включите файлы, которые вам нужны в одном бандле в main.js

```
// File: main.js

// Импортируем работу с вендорами
import './vendors.js'
```

Target (Цель) JavaScript ныне работает как на клиентской стороне, так и на стороне сервера. Мы можем создавать множественные цели, чтобы решать такие проблемы:

```
const path = require('path');
const serverConfig = {
  target: 'node',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'main.node.js',
  },
  // ...
};

const clientConfig = {
  target: 'web',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'main.js',
  },
  // ...
};
```

```
// ...  
};  
  
module.exports = [serverConfig, clientConfig];
```

Однако опыт показывает что так делать не всегда удобно, а иногда и вовсе нежелательно. Держать все конфигурации в одном файле - нехорошая идея, лучше создать две директории и два независимых webpack.config.js.

target - данное свойство указывает цель для чего мы создаем бандл, это может быть:

- async-node - бандл для асинхронной ноды, который будет тянуть модули с промисами
- node - бандл для синхронной ноды
- electron-renderer - бандл для рендер-процесса в Electron
- electron-preload - бандл для прелоудера в Electron
- web - бандл для браузера. Данный таргет является дефолтным.
- webworker - бандл для воркера. В зависимости от выбора бандла Webpack будет по-разному импортировать модули и обращаться с ними. **Кэширование** можем кэшировать неизменные части нашего приложения, для того чтобы Webpack быстрее собирал наше приложение. По умолчанию файлы кэшируются в памяти в mode: development и не кэшируются вовсе, если mode: production. Для того чтобы Webpack кэшировал все не в оперативной памяти, а в постоянной — достаточно просто указать следующее свойство:

```
module.exports = {  
  cache: {  
    type: 'filesystem', // По умолчанию 'memory'  
  },  
};
```

Webpack будет кэшировать билд в node_modules/.cache/webpack и автоматом пропускать то, что не изменилось! Директорию для кэша можно изменить с помощью cacheDirectory:


```
const path = require('path');  
module.exports = {  
  cache: {  
    type: 'filesystem', // По умолчанию 'memory'  
  
    // Устанавливаем директорию для кэша  
    cacheDirectory: path.resolve(__dirname, '.temporary_cache')  
  }  
};
```

```
},  
};
```

Для того чтобы отключить кэш вовсе достаточно просто добавить `cache: false`, интересным замечанием будет то, что `cache: true` — то же самое, что и `cache: {type: 'memory'}` **Watch (Наблюдение за изменениями файлов)** При изменении файлов запускается пересборка проекта. Это очень полезно при разработке. Для того чтобы Webpack "подсматривал" за нашими файлами достаточно просто указать:

```
module.exports = {  
  //...  
  watch: true,  
  
  // Настройки для watch  
  watchOptions: {  
  
    // Директории, которые watch будет игнорировать  
    ignored: [/node_modules/]  
  }  
};
```

****DevServer**** Webpack ещё и умеет запускать свой http-сервер, для того чтобы у вас была live-reload (перезагрузка при рекомпиляции)

```
const path = require('path');  
  
module.exports = {  
  
  // Конфигурация для нашего сервера  
  devServer: {  
  
    // Здесь указывается вся статика, которая будет на нашем сервере  
    static: {  
      directory: path.join(__dirname, 'public'),  
    },  
  
    // Сжимать ли бандл при компиляции   
    compress: true,
```

```
// Порт на котором будет наш сервер
port: 9000,
},
};
```

Чтобы Webpack показывал нам ошибки, если вдруг что-то пошло не так, прогресс-бар при компиляции:

```
module.exports = {
  //...
  devServer: {
    // ...
    client: {

      // Показывает ошибки при компиляции в самом браузере
      overlay: {

        // Ошибки
        errors: true,

        // Предупреждения
        warnings: false,
      },

      // Показывает прогресс компиляции
      progress: true
    },
  },
};
```

У самого сервера есть безграничные возможности: от поддержки Hot Module Replacement до поддержки WS-сервера и HTTPS соединения, все их можно посмотреть

- <https://webpack.js.org/configuration/dev-server/> **Пример на TypeScript** Написать конфигурацию для TypeScript с подтягиванием файлов [.ts, .tsx, .js], картами для исходников

```
// Подтягиваем модуль для удобной работы с путями
const path = require('path');
```

```

module.exports = {

  // Точка входа
  entry: './src/index.ts',
  mode: 'development'

  // Говорим, что нам нужна карта исходников 🗺️
  devtool: 'inline-source-map',
  module: {
    rules: [
      // Компилируем TypeScript
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      },
    ],
  },

  // Говорим что если не указано расширение файла, то пробуем эти варианты
  // @see https://webpack.js.org/configuration/resolve/#resolveextensions
  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },

  // Точка выхода
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};

```

7. Create React App

(<https://create-react-app.dev/docs/getting-started>) отличный инструмент для быстрого старта React-приложений. Не тратите время на настройку Webpack, Babel и других привычных инструментов. Они заранее настроены и спрятаны, так что разработчики могут сфокусироваться на коде и бизнес-логике приложения.

```

npm create-react-app my-app

```



```
npm create react-app my-app
cd my-app/
npm start
```

Команда `npm run eject` копирует все конфиги и транзитивные зависимости (Webpack, Babel, ESLint и т. д.) в ваш проект, чтобы вы могли их контролировать. **Стек разработки фиксированный и стабильный** В релизах Create React App присутствуют только stage-3 функции (release candidate)

cross-env Советую использовать `cross-env`, чтобы кроссплатформенно устанавливать переменные окружения.

```
"scripts": {
  "start": "cross-env NODE_PATH=src/scripts react-scripts start"
}
```

используете `.env`-конфиг, просто добавьте переменную `NODE_PATH`:

```
NODE_PATH=src/scripts
```

Переменные окружения Create React App по умолчанию поддерживает `.env` (используя `dotenv`-пакет). Просто создайте в корневой папке `.env` и запустите приложение. Не забудьте добавить каждой переменной префикс `REACT_APP_`. Больше информации — в официальной документации, в разделах Adding Custom Environment Variables (<https://github.com/facebook/create-react-app/blob/main/packages/react-scripts/template/README.md#adding-custom-environment-variables>) и Adding Development Environment Variables In `.env`. (<https://github.com/facebook/create-react-app/blob/main/packages/react-scripts/template/README.md#adding-development-environment-variables-in-env>) **Поддержка нескольких `.env`-конфигов** Иногда полезно разделить конфиги по типу окружения (`dev/test/prod`). Например, вот `.env.development`

```
REACT_APP_GOOGLE_CLIENT_ID = XXX-YYY-ZZZ.apps.googleusercontent.com
REACT_APP_API_PROTOCOL = http:
REACT_APP_API_HOST = localhost:3000
NODE_PATH = src/scripts
PORT = 9001
```

А вот `.env.production`:

```
REACT_APP_GOOGLE_CLIENT_ID = ZZZ-YYY-XXX.apps.googleusercontent.com
REACT_APP_API_PROTOCOL = https:
REACT_APP_API_HOST = api.my-application.com
NODE_PATH = src/scripts
```

Сейчас это можно сделать, установив dotenv и обновив npm scripts:

```
"scripts": {
  "start": "node -r dotenv/config ./node_modules/.bin/react-scripts start
dotenv_config_path=.env.development",
  "build": "node -r dotenv/config ./node_modules/.bin/react-scripts build
dotenv_config_path=.env.production"
}
```

.env*-конфиги:

- .env — стандартный (общий) конфиг;
- .env.development, .env.test, .env.production — в зависимости от окружения;
- .env.local — локальный конфиг для переопределения любых переменных в зависимости от окружения разработчика. Игнорируется системой контроля версий;
- .env.development.local, .env.test.local,
- .env.production.local — локальный конфиг в зависимости от окружения. Игнорируется системой контроля версий. Приоритет конфигов (шаг пропускается, если файл конфига не существует):
- npm test — .env.test.local, env.test, .env.local, .env;
- npm run build — .env.production.local, env.production, .env.local, .env;
- npm start — .env.development.local, env.development, .env.local, .env. **Изменить порт dev-сервера** Добавьте переменную окружения PORT с помощью cross-env:

```
"scripts": {
  "start": "cross-env PORT=9001 react-scripts start"
}
```

или .env-конфига:

```
PORT=9001
```

Дополнительная конфигурация - список возможных настроек с помощью переменных окружения: BROWSER - Create React App открывает браузер, настроенный по умолчанию, но можно задать определенный браузер или установить none, чтобы отключить эту фичу. Также можно указать на Node.JS скрипт, который будет выполняться каждый раз при старте dev-сервера. HOST - По умолчанию — localhost. PORT - По умолчанию — 3000. Если он занят, Create React App предложит запустить приложение на следующем доступном порте. Или можно задать определенный порт. HTTPS - Если установлено в true — локальный dev-сервер будет запущен в https-режиме. PUBLIC_URL - Обычно Create React App ожидает, что приложение расположено в корне веб-сервера или путь определен в package.json (homepage). Можно переопределить ссылку для всех ассетов. Это полезно, если вы используете CDN для хостинга приложения. CI - Если установлено в true, Create React App будет обрабатывать warnings как ошибки. Запускает тесты без — watch параметра.

8. Readme.md

Используется в проектах для описания и порядка запуска. Часто оформляется с применением разметки Markdown - <https://docs.github.com/ru/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax>