НЕТОЛОГИЯ

РАБОЧЕЕ ОКРУЖЕНИЕ

ДЕНИС ЕЖКОВ

ДЕНИС ЕЖКОВ

Frontend-разработчик в «Ростелеком IT»

aka_SKIff

facebook.com/ezhkov

@ezhkov

ПЛАН ЗАНЯТИЯ

1. Рабочее окружение
2. npm
3. live-server
4. ESLint
5. Babel

РАБОЧЕЕ ОКРУЖЕНИЕ

РАБОЧЕЕ ОКРУЖЕНИЕ

Под рабочим окружением мы будем понимать не только (и не столько) настройку редактора кода или IDE, сколько использование инструментов, не завязанных на IDE и облегчающих:

- разработку
- тестирование
- соблюдение правила

IDE И РЕДАКТОР КОДА

Для большинства программистов выбор IDE является сугубо личным делом, поэтому мы будем настраивать рабочее окружение независимо от IDE.

HTTP-СЕРВЕР

Задача: установить локальный HTTP-сервер для разработки, тестирования и отладки приложения.

Очевидное решение: воспользоваться Google и найти несколько решений, в том числе сложные Apache или Nginx.

ПРИЧИНЫ ИСПОЛЬЗОВАТЬ HTTP-СЕРВЕР

1. Повышается скорость разработки из-за использования профессиональных инструментов
2. Разработка в команде требует знания и умения работать с общепринятыми инструментами
3. Некоторые приложения невозможно протестировать оффлайн, без HTTP-сервера

NPM

npm — менеджер пакетов для JS, реестр готовых пакетов и утилита командной строки в одном флаконе.

Стандарт де-факто в мире JS для организации рабочего окружения.

На сегодняшний день содержит более 800 000 готовых пакетов.

«МАГАЗИН» ПАКЕТОВ

Для iPhone есть AppStore, для Android — Google Play, а для JS — npm.

УСТАНОВКА

npm уже входит в состав Node.js*.

*Должен быть установлен на ваш ПК перед началом курса.

Список рекомендуемого ПО для курса.

ПРОВЕРЯЕМ NPM

Чтобы проверить, что npm действительно установлен, выполните команду в терминале:

```
$ npm -v
```

Версия npm должна быть 6.5.0 или выше.

ОБНОВЛЯЕМ NPM

Если версия npm ниже 6.5.0, то потребуется обновить менеджер пакетов. Для этого в терминале выполните команду:

```
$ npm install --global npm
```

РАБОТА С ТЕРМИНАЛОМ

Краткое руководство по работе с терминалом - необходимо изучить самостоятельно.

ТЕРМИНАЛ В VSCODE

В VSCode уже встроен терминал, можно его открыть сочетанием клавиш `Ctrl` + `` ` `` (Windows/Linux), `^` + `` ` `` (Mac).

УСТАНАВЛИВАЕМ HTTP-СЕРВЕР

Установим пакет http-server с использованием npm:

```
$ npm install --global http-server
```

КОМАНДА ДЛЯ ЛЮБОГО ПАКЕТА

Команда `npm install [--global] <package-name>` позволяет нам устаналивать любой пакет из реестра https://npmjs.com.

ФЛАГ `--global`

Флаг `--global` указывает, что пакет устанавливается глобально (global mode), тоесть будет доступен в рамках всей системы (или текущего пользователя ОС).

СПРАВКА NPM

Если вам нужна помощь при работе с npm, то вы можете набрать команды помощи:

```
$ npm help
$ npm help <command>
$ npm <command> --help
```

Полный перечень всех команд с описанием представлен на веб-странице: https://docs.npmjs.com/cli-documentation/

ЗАПУСКАЕМ HTTP-СЕРВЕР

Откроем в терминале каталог нашего проекта и запустим HTTP-сервер:

```
$ cd src
$ http-server
Starting up http-server, serving ./
Available on:
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

Остаётся открыть браузер с адресом http://localhost:8080

СПРАВКА

Большинство пакетов, поставляемых через npm, содержат встроенную справку, которую можно вызвать командой:

```
$ <package-name> --help
```

Например, для пакета http-server команда будет выглядеть так:

```
$ http-server --help
```

РЕПОЗИТОРИЙ ПАКЕТА

Одной командой мы получили готовый и легковесный HTTP-сервер с большим набором опций.

Кроме того, мы можем даже сразу открыть страницу GitHub-проекта:

```
$ npm repo http-server
```

Репозиторий проекта очень полезен, например, если нужно понять, как бороться с тем или иным багом.

МИНУСЫ

Минусы такого подхода:

- засоряется «глобальная область видимости» (global mode)
- при работе в команде придётся писать всем инструкции

Можно ли настраивать и устанавливать пакеты только для текущего проекта?

ПАКЕТ

В терминах npm пакет - это любой каталог*, содержащий файл `package.json`.

Для пакета можно настроить зависимости — пакеты, которые нужны для функционирования (зависимости) или для разработки, тестирования (dev-зависимости).

Кроме того, можно вынести ключевые команды в скрипты.

* Более подробно

ИНИЦИАЛИЗИРУЕМ ПАКЕТ

Выполним в нашем проекте (не в каталоге `src`, а в корне нашего проекта):

```
$ npm init
```

Запустится конфигуратор пакета, который задаст вам ряд вопросов, после чего сгенерируется файл `package.json`.

WARNING!

Важно: старайтесь создавать пакеты в каталогах, которые не содержат в пути пробелов, спец.символов, кириллицы.

КЛЮЧЕВЫЕ ПОЛЯ `package.json`

- `name` — название пакета
- `version` — версия (в соответствии с https://semver.org)
- `scripts` — скрипты (удобные сокращения для команд)
- *`dependencies` — зависимости, необходимые для функционирования пакета
- *`devDependencies` — зависимости, необходимые для разработки и (или) тестирования пакета

* Разделы, которых у нас пока нет.

УСТАНОВИМ ПАКЕТ ЛОКАЛЬНО

Зависимости устанавливаются с помощью команды `npm install`:

```
$ npm install [--save-dev] <package-name>
```

ФЛАГ `--save-dev`

Флаг `--save-dev` означает, что зависимость нужна только для разработки и тестирования. Поскольку пакет http-server нам нужен как раз для этих целей, то он является dev-зависимостью:

```
$ npm install --save-dev http-server
```

DEV DEPENDENCIES

В результате выполнения команды в нашем проекте создался каталог `node_modules` и в `package.json` появилась секция `devDependencies`:

```
"devDependencies": {
  "http-server": "^0.11.1"
}
```

ПОСМОТРИМ СПИСОК ПАКЕТОВ

Посмотреть список локально установленных пакетов мы можем с помощью команды:

```
$ npm list
```

СКРИПТ ДЛЯ СЕРВЕРА

Теперь возникает вопрос, как же запустить http-server?

Сам исполняемый файл хранится в каталоге `node_modules/.bin/http-server`. Чтобы его запустить, мы можем прописать его в секцию `scripts` следующим образом:

```
"scripts": {
  "serve": "http-server src",
  "test": "echo \"Error: no test specified\" && exit 1"
}
```

И в командной строке просто запускать команду:

```
$ npm run serve
```

SCRIPTS

В скрипты можно прописывать любые команды, которые могут выполниться в терминале. Таким образом, это удобный способ:

- дать псевдоним длинной команде
- определить набор стандартных команд для вашего проекта

СТАНДАРТНЫЕ ИМЕНА

Ряд имён в скриптах стандартизирован и позволяет опускать ключевое слово `run`. Например, изменим секцию `scripts`:

```
"scripts": {
  "start": "http-server src",
  "test": "echo \"Error: no test specified\" && exit 1"
}
```

Теперь для запуска сервера нам достаточно написать команду:

```
$ npm start
```

СПИСОК СТАНДАРТНЫХ ИМЁН

- `start`
- `stop`
- `test`
- `restart`

Полный список приведён на странице
https://docs.npmjs.com/misc/scripts.html#description

ПОИСК ПАКЕТА

С http-server всё достаточно хорошо, но для разработки нужен более мощный инструмент, который бы содержал в себе функцию автоматической перезагрузки страницы при изменении любого файла.

Это называется **live reload**.

В ПОИСКАХ LIVE RELOAD

Поищем такой инструмент через npm:

```
$ npm search --long live reload
```

Какую команду нужно использовать для установки?

LIVE-SERVER

УСТАНОВИМ LIVE-SERVER

Среди прочих мы увидим live-server. Установим его командой:

```
$ npm install --save-dev live-server
```

ЗАМЕНА В `scripts`

И заменим http-server на live-server в скрипте `start`.

```
"scripts": {
  "start": "live-server src",
  "test": "echo \"Error: no test specified\" && exit 1"
}
```

Теперь любое изменение файлов в каталоге `src` приводит к перезагрузке страницы в браузере, что достаточно удобно.

УДАЛЕНЯЕМ НЕНУЖНЫЕ ПАКЕТЫ

Удалим глобально установленный http-server:

```
$ npm uninstall --global http-server
$ npm list --global
```

Удалим локально установленный http-server:

```
$ npm unistall http-server
$ npm list
```

ЧТО ХРАНИМ В GIT?

Возникает вопрос, какие из созданных npm файлов и каталогов стоит хранить в Git?

Храним в Git:

- `package.json`;
- `package-lock.json`.

Не храним в Git: `node_modules`

УСТАНАВЛИВАЕМ ЗАВИСИМОСТИ

Все зависимости из `node_modules` легко восстанавливаются командой:

```
$ npm install
```

`.gitignore`

Лучше использовать готовый `.gitignore`, собранный сообществом:
https://github.com/github/gitignore/blob/master/Node.gitignore

КОМАНДНАЯ РАЗРАБОТКА

Стиль кодирования — одна из самых больных тем при разработке проектов. Здесь всё так же, как с IDE — у каждого свой вкус, но если мы хотим получить хороший продукт, нужно установить правила и придерживаться их.

Посмотрим, помогут ли инструменты, распространяемые через npm нам чем-то помочь.

ESLINT

ESLINT

ESLint — представитель инструментов, отслеживающих стиль кодирования и типичные ошибки в мире JS.

УСТАНАВЛИВАЕМ ESLINT

Установим его с помощью npm:

```
$ npm install --save-dev eslint
```

ДЕМО

Посмотреть на работу ESLint в live-режиме: https://eslint.org/demo/

NPX

npm включает в себя утилиту npx, позволяющую запускать команды из `node_modules/.bin` либо выполняя временную установку.

ФОРМИРУЕМ НАБОР ПРАВИЛ

Поскольку набор правил мы формируем только один раз, воспользуемся ею:

```
$ npx eslint --init
```

КОНФИГУРАТОР

Будет запущен конфигуратор, в котором необходимо выбрать:

- How would you like to use ESLint? To check syntax, find problems, and enforce code style
- What type of modules does your project use? JavaScript modules (import/export)
- Which framework does your project use? None of this
- Where does your code run? Browser
- How would you like to define a style for your project? Use a popular style guide
- Which style guide do you want to follow? Airbnb
- What format do you want your config file to be in? JSON
- Would you like to install them now with npm? Y

AIRBNB STYLEGUIDE

Изучите самостоятельно Styleguide Airbnb.

В данном Styleguide описан не только стиль кодирования, но и причины, по которым выбраны те или иные подходы.

.eslintrc.json

ESLint будет хранить свои настройки в файле `.eslintrc.json`:

```
{
    "env": {
        "browser": true,
        "es6": true
    },
    "extends": "airbnb-base",
    "globals": {
        "Atomics": "readonly",
        "SharedArrayBuffer": "readonly"
    },
    "parserOptions": {
        "ecmaVersion": 2018,
        "sourceType": "module"
    },
    "rules": {
    }
}
```

`.eslintrc.json`

Мы можем изменить, дополнить правила с помощью секции `rules`.

Для лекций нам потребуется:

```
"rules": {
    "no-restricted-syntax": [
        "error",
        "LabeledStatement",
        "WithStatement"
    ]
}
```

LINT SCRIPT

Создадим отдельный скрипт `lint`, который и будет запускать ESLint в соответствии с установленными настройками:

```
"scripts": {
  "start": "live-server src",
  "test": "echo \"Error: no test specified\" && exit 1",
  "lint": "eslint ."
},
```

ЗАПУСКАЕМ ЛИНТЕР

Поскольку имя `lint` не относится к стандартным, нам необходимо будет запускать его следующим образом:

```
$ npm run lint
```

РЕЗУЛЬТАТ РАБОТЫ ЛИНТЕРА

```
  1:1   warning  Unexpected console statement                  no-console
  1:21  error    Newline required at end of file but not found  eol-last
  1:21  error    Missing semicolon                              semi

✖ 3 problems (2 errors, 1 warning)
  2 errors and 0 warnings potentially fixable with the `--fix` option.
```

FIX ОТ ESLINT

ESLint предлагает нам передать ключ `--fix` для того, чтобы исправить проблемы. Чтобы в npm-скрипт передать этот флаг, нам нужно воспользоваться специальным синтаксисом:

```
$ npm run lint -- --fix
```

Флаги команд следует передавать после `--`.

FIX OT ESLINT

Мы рекомендуем исправить ошибки и warning'и ESLint'а вручную, сразу во время написания кода.

См. информацию про плагин далее.

ИТОГ

С помощью инструмента ESLint мы получили возможность определять стиль кодирования для проекта и проверять его с помощью npm.

Но как сделать так, чтобы ошибки стиля автоматически подсвечивались в редакторе кода?

СТАВИМ ПЛАГИН ДЛЯ РЕДАКТОРА

Придётся устанавливать плагин для редактора. Для VSCode необходимо зайти на вкладку расширений и установить плагин **ESLint** от *Dirk Baeumer*. После этого нам прямо в редакторе будут подсвечиваться подсказки от ESLint в соответствии с установленной конфигурацией.

Для других редакторов/IDE — аналогично.

BABEL

ПОДДЕРЖКА СТАНДАРТОВ ECMASCRIPT

Представим ситуацию: мы пишем библиотеку, которую хотим использовать в нескольких разных проектах.

Как нам в этом случае поступить с поддержкой стандартов различных версий?

ВОЗМОЖНЫЕ РЕШЕНИЯ

1. Выбрать самый старый и писать на нём (тогда не получится использовать новые возможности)
2. Выбрать самый новый и не думать о совместимости (тогда придётся переписывать)
3. Одновременно поддерживать несколько версий (трудоёмко)

BABEL

Проект Babel предлагает альтернативное решение: транспайлер — инструмент, преобразующий код из одной версии стандарта в другую, более старую*.

То есть мы можем использовать большинство** новых возможностей языка и получать код, написанный в режиме совместимости с нужной нам версией.

* Это достаточно упрощённое описание транспайлера, но оно помогает понять суть.

** Если они поддерживаются транспайлером.

УСТАНАВЛИВАЕМ BABEL

```
$ npm install --save-dev @babel/core @babel/cli @babel/preset-env
$ npm install @babel/polyfill core-js@3
```

Обратите внимание: `@babel/polyfill` и `core-js@3` - это не dev-зависимость.

ДЕМО

Посмотреть на работу Babel в live-режиме: https://babeljs.io/en/repl.html

ФОРМИРОВАНИЕ НАБОРА ПРАВИЛ

В отличие от ESLint, для Babel конфигурационный файл необходимо создать вручную (`.babelrc`):

```
{
  "presets": [
    [
      "@babel/preset-env", {
        "useBuiltIns": "usage",
        "corejs": 3
      }
    ]
  ]
}
```

ДОБАВЛЯЕМ В `scripts`

После чего в `scripts` прописать:

```
"scripts": {
  "start": "live-server src",
  "test": "echo \"Error: no test specified\" && exit 1",
  "lint": "eslint .",
  "build": "babel src -d dist"
},
```

ЗАПУСКАЕМ ТРАНСПАЙЛИНГ

```
$ npm run build
```

В результате выполнения этой команды создастся файл `dist/js/index.js`, в котором будет только совместимый с текущей средой выполнения и настройками авторов Babel'а код.

Preset - это сформированный набор плагинов, собранный под одним именем.

POLYFILL

Polyfill от Babel - это реализация функциональности, которая отсутствовала в предыдущих версиях ES.

`"useBuiltIns": "usage"` говорит о том, что Babel будет самостоятельно подключать полифиллы на основе анализа используемых вами возможностей.

BROWSERSLIST

Файл с именем `.browserslistrc` — позволяет установить, поддержку каких браузеров (окружений) необходимо обеспечивать, исходя из статистики caniuse.com

```
# Browsers that we support

last 1 version
> 1%
maintained node versions
not dead
```

BABEL & ESLINT

ESLint на данный момент анализирует все файлы в нашем проекте (включая те, что в каталоге `dist`).

Изменим настройки ESLint так, чтобы каталог `dist` игнорировался полностью. Для этого нам нужен файл `.eslintignore`:

```
dist
```

Таким образом, ESLint будет игнорировать данный каталог, как и плагины для редакторов/IDE.

ЗАЧЕМ ЭТО ВСЁ?

Это инфраструктура современного JS.

В дальнейшем, с каким бы вы проектом или фреймворком (Angular, Vue, React) не сталкивались, с большой долей вероятности он будет использовать эту инфраструктуру.

ПОДДЕРЖКА ЭКСПЕРИМЕНТАЛЬНЫХ ВОЗМОЖНОСТЕЙ

Babel с помощью плагинов поддерживает то, чего ещё нет в стандарте, но что вы можете использовать уже сейчас.

Список плагинов, поддерживающих экспериментальные возможности: Experimental Plugins.

OPTIONAL CHAINING

Рассмотрим одну из таких возможностей - Optional Chaining:

```
const obj = {
  prop: {
    value: 999,
  },
};

const existent = obj?.prop?.value; // 999
// obj.data.value: Uncaught TypeError: Cannot read property 'value' of undefined
const notExistent = obj?.data?.value; // undefined вместо ошибки
```

OPTIONAL CHAINING

Установим поддержку в Babel:

```
$ npm install --save-dev @babel/plugin-proposal-optional-chaining
```

.babelrc:

```
{
  "plugins": ["@babel/plugin-proposal-optional-chaining"]
}
```

После этого код с предыдущего слайда будет компилироваться и работать.

Проверку ESLint он не пройдёт, для этого необходима дополнительная настройка.

ВАЖНО

Очень осторожно относитесь к использованию экспериментальных возможностей, ещё не принятых в стандарт, т.к. вполне возможно, что в стандарт они так и не попадут, и Babel прекратит поддерживать плагины, которые осуществляли транспайлинг.

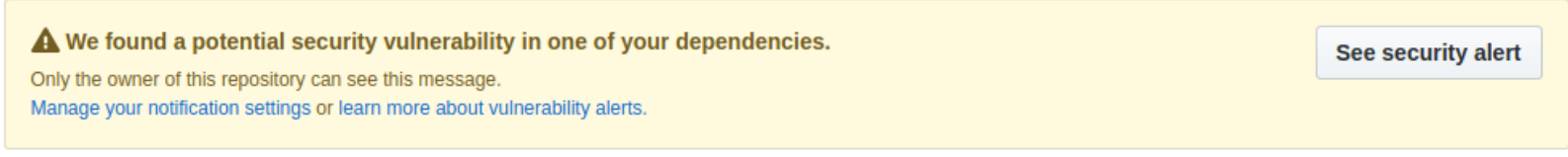
Мы показали вам это лишь для того, чтобы вы понимали, как в некоторых проектах используются возможности, поддержки которых нет в браузере и в стандарте.

УЯЗВИМОСТИ

УЯЗВИМОСТИ

Тема наличия уязвимостей в используемых инструментах и библиотеках является достаточно важной.

Так, если вы используете подверженные уязвимостям версии, тот же GitHub будет вам выдавать предупреждение:



Проверить наличие уязвимых пакетов вы можете с помощью команды: `npm audit`, после чего принимать решение об обновлении с помощью `npm audit fix`.

Смотрите детали в документации.

УЯЗВИМОСТИ

```
$ npm audit

=== npm audit security report ===

# Run  npm install forever@1.0.0  to resolve 1 vulnerability
SEMVER WARNING: Recommended action is a potentially breaking change
┌───────────────┬──────────────────────────────────────────────────────────────┐
│ Low           │ Regular Expression Denial of Service                         │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ braces                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ forever                                                      │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ forever > forever-monitor > chokidar > anymatch > micromatch │
│               │ > braces                                                     │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/786                             │
└───────────────┴──────────────────────────────────────────────────────────────┘
```

ИТОГИ

ИТОГИ

1. npm — позволяет упростить работу с проектами и зависимостями
2. live-server — позволяет упростить разработку и отладку в браузере
3. ESLint — позволяет искать ошибки и соблюдать стиль кодирования
4. Babel — решает проблему поддержки предыдущих версий стандарта

ТРЕБОВАНИЯ К ДОМАШНИМ РАБОТАМ

Начиная с сегодняшнего дня во всех домашних заданиях будет требоваться:

- использования npm при формировании проекта ДЗ
- соответсвия ESLint набору правил Airbnb на уровне отсутствия ошибок (error - **не допускаются**, warning - допускаются)

ССЫЛКИ НА ДОКУМЕНТАЦИЮ

- npm
- live-server
- ESLint
- Babel
- Browserslist

НЕТОЛОГИЯ

Задавайте вопросы и напишите отзыв о лекции!

ДЕНИС ЕЖКОВ


aka_SKIff


facebook.com/ezhkov


@ezhkov