***NodeJS*** – это среда, которая позволяет запускать программы на JavaScript не в вашем браузере, а на вашем компьютере. Скачать можно на <https://nodejs.org/>

***NPM*** – это менеджер пакетов для *NodeJS* и устанавливается в комплекте с ним.

Исходники проекта рекомендовано хранить в папке ***src***. Конечная сборка будет находиться в папке ***dist***.

Для установки зависимостей, необходимых только при разработке, используется флаг ***--save-dev***.

*npm install prettier --save-dev*

В файле конфигурации ***package.json*** он отобразится как ***devDependencies***.

*"devDependencies": {*

*"prettier": "^2.8.0",*

*}*

Рассмотрим пример настройки автоматизации проверок внешнего вида кода с помощью пакетов *NodeJS*:

1. Перейдем в папку проекта и создадим новый проект командой

*npm init*

1. Добавим в исключения *.gitignore* папку ***node\_modules/***
2. Установим пакеты ***Prettier*** и ***Stylelint***.

*npm i prettier*

*npm i stylelint*

1. Скопируем конфигурационный файл ***.prettierrc*** вместе с содержимым.
2. Скопируем конфигурационный файл ***stylelint.config.js*** вместе с содержимым.
3. Скачиваем плагин ***Stylelint*** для *VSCode*.
4. Добавляем в файл ***package.json*** блок со скриптами

*"scripts": {*

*"test": "echo 'Привет!'",*

*"check": "prettier --check './src/\*\*/\*.{html,css,js}'",*

*"format": "prettier --write './src/\*\*/\*.{html,css,js}'",*

*"lint": "stylelint './src/\*\*/\*.css'",*

*"lintfix": "stylelint './src/\*\*/\*.css' --fix",*

*"checkAll": "npm run check && npm run lint",*

*"fixAll": "npm run format && npm run lintfix"*

*},*

Теперь можно запускать в терминале команды:

*npm run checkAll // проверить весь код обоими плагинами*

*npm run fixAll // исправить весь код обоими плагинами*

Самый популярный сборщик ***Webpack***, но он требует знаний *JavaScript*.

Рассмотрим сборщик ***Gulp***.

1. Устанавливаем сборщик на компьютер.

*npm install -g gulp*

1. Устанавливаем сборщик внутрь нужного проекта.

*npm install gulp --save-dev*

1. Создаем файл конфигурации ***gulpfile.js*** в корне проекта и подключаем сборщик.

*const gulp = require('gulp');*

1. Устанавливаем вспомогательные плагины.

*npm i browser-sync --save-dev // сервер просмотра в реальном времени*

*npm i del@6.0.0 --save-dev // удаляет файлы из сборки при удалении из исходников*

*npm i gulp-concat-css --save-dev // склеивает CSS-файлы в один*

*npm i gulp-plumber --save-dev // игнорирует ошибки при сборке*

1. Подключаем плагины в файл ***gulpfile.js***

*const concat = require('gulp-concat-css');*

*const plumber = require('gulp-plumber');*

*const del = require('del');*

*const browserSync = require('browser-sync').create();*

1. В файле ***gulpfile.js*** создаем функцию для копирования всех *html*-файлов без изменений.

*function html() {*

*return gulp.src('src/\*\*/\*.html') // берем все html-файлы из папки src*

*.pipe(plumber()) // игнорируем ошибки*

*.pipe(gulp.dest('dist/')) // перемещаем их в папку dist*

*}*

*exports.html = html // экспортируем функцию под именем html*

1. В файле ***gulpfile.js*** создаем функцию для склеивания все *css*-файлов в один итоговый *css*-файл.

*function css() {*

*return gulp.src('src/blocks/\*\*/\*.css') // берем все css-файлы из папки src/blocks*

*.pipe(plumber()) // игнорируем ошибки*

*.pipe(concat('bundle.css')) // склеиваем их в bundle.css*

*.pipe(gulp.dest('dist/')) // перемещаем их в папку dist*

*}*

*exports.css = css; // экспортируем функцию под именем css*

1. В файле ***gulpfile.js*** создаем функцию для копирования всех файлов изображений без изменений.

*function images() {*

*return gulp.src('src/images/\*\*/\*.{jpg,png,svg,gif,ico,webp,avif}') // берем все картинки из src/images*

*.pipe(gulp.dest('dist/images')) // перемещаем их в папку dist*

*}*

*exports.images = images; // экспортируем функцию под именем images*

1. Повторить по аналогии для шрифтов и видео.
2. В файле ***gulpfile.js*** создаем функцию для очистки папки *dist* перед каждой сборкой.

*function clean() {*

*return del('dist'); // удаляем папку dist*

*}*

*exports.clean = clean; // экспортируем функцию под именем clean*

1. В файле ***gulpfile.js*** создаем функцию для сборки проекта в рамках одной команды. Функция ***series()*** выполняет команды по очереди, а функция ***parallel()*** выполняет команды параллельно.

*const build = gulp.series(clean, gulp.parallel(html, css, images));*

*exports.build = build; // экспортируем функцию под именем build*

1. В файле ***gulpfile.js*** создаем функцию для отслеживания изменений в файлах папки *src* и автоматической пересборкой проекта.

*function watchFiles() {*

*gulp.watch(['src/\*\*/\*.html'], html);*

*gulp.watch(['src/blocks/\*\*/\*.css'], css);*

*gulp.watch(['src/images/\*\*/\*.{jpg,png,svg,gif,ico,webp,avif}'], images);*

*}*

*const watchapp = gulp.parallel(build, watchFiles);*

*exports.watchapp = watchapp; // экспортируем функцию под именем watchapp*

1. В файле ***gulpfile.js*** создаем функцию для запуска сервера в реальном времени.

*function serve() {*

*browserSync.init({ // стартуем сервер*

*server: {*

*baseDir: './dist' // на основе данных из dist*

*}*

*});*

*}*

1. В файле ***gulpfile.js*** добавляем пайпы для перезапуска сервера при изменении файлов *html*, *css* и изображений.

*function html() {*

*return gulp.src('src/\*\*/\*.html')*

*.pipe(plumber())*

*.pipe(gulp.dest('dist/'))*

*.pipe(browserSync.reload({stream: true})); // перезапуск сервера*

*}*

1. В файле ***gulpfile.js*** добавляем функцию сервера в список задач отслеживания.

*const watchapp = parallel(build, watchFiles, serve);*

1. В файле ***gulpfile.js*** задаем функцию по умолчанию.

*exports.default = watchapp; // стартует по умолчанию при команде “gulp”*

1. В файле ***package.json*** в зону *scripts* добавляем команды для запуска сборки проекта и режима разработки.

*"scripts": {*

*"build": "gulp build",*

*"dev": "gulp"*

*}*

Собрать итоговый проект можно командой:

*npm run build*

Запустить проект в режиме разработки можно командой:

*npm run dev*

Опубликовать сайт бесплатно можно на многих хостингах. Например, на [Heroku](https://www.heroku.com/), [Netlify](https://www.netlify.com/) и [Vercel](https://vercel.com/).

Рассмотрим пример публикации на ***Netlify***.

1. Создаем публичный репозиторий в *GitHub* и заливаем туда свой проект.
2. В файле ***package.json*** в зону *scripts* добавляем команду для фиксирования изменений и заливки в *GitHub*. Если исправление или форматирование не удастся, то код не отправится в *GitHub*. В итоге он будет выглядеть так:

*"scripts": {*

*"test": "echo 'Привет!'",*

*"check": "prettier --check './src/\*\*/\*.{html,css,js}'",*

*"format": "prettier --write './src/\*\*/\*.{html,css,js}'",*

*"lint": "stylelint './src/\*\*/\*.css'",*

*"lintfix": "stylelint './src/\*\*/\*.css' --fix",*

*"checkAll": "npm run check && npm run lint",*

*"fixAll": "npm run format && npm run lintfix",*

*"build": "gulp build",*

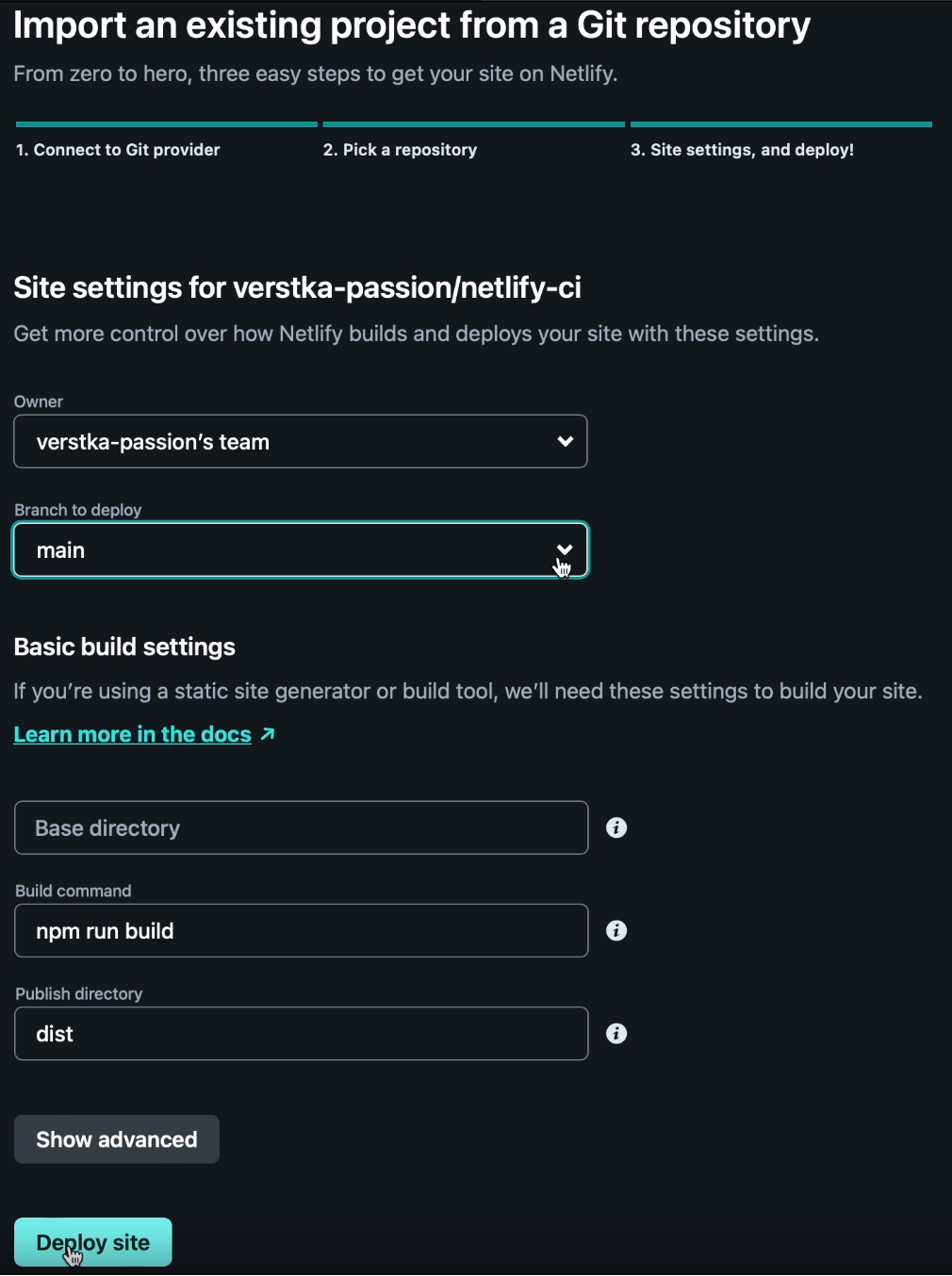
*"dev": "gulp",*

*"fixAndBuild": "npm run fixAll && npm run build",*

*"lintfixAndPush": "npm run fixAll && git add . && git commit -m 'подготовили к публикации' && git push"*

*}*

1. Авторизовываемся на *Netlify* через *GitHub*.
2. Выбираем импорт из GitHub, репозиторий, ветку, команду для сборки и имя папки с результатом.



1. Теперь любые изменения, опубликованные командой *lintfixAndPush*, будут автоматически залиты а хостинг.

**Постпроцессинг** – это преобразование уже написанного кода под специфические требования браузеров.

Для описания в инфраструктуре целевых браузеров, создана специальная утилита [*Browserlist*](https://browsersl.ist/). Рекомендуется поддержка **пяти последних версий** браузеров.

Пример конфигурации в файле ***package.json***:

*"browserslist": [*

*"last 2 versions",*

*"not dead",*

*"not ie <= 11",*

*"not op\_mini all"*

*]*

Для постпроцессинга *CSS* создана утилита [PostCSS](https://postcss.org/) и множество плагинов для нее.

Плагин ***autoprefixer*** смотрит в настройки *browserlist* внутри *package.json* и подставляет вендорные префиксы для браузеров, которым они нужны. Для его подключения нужно:

1. Установить зависимость в проект.

*npm i autoprefixer --save-dev*

1. В файле ***gulpfile.js*** подключаем плагин.

*const autoprefixer = require('autoprefixer');*

1. Добавляем его в хранилище плагинов.

*const plugins = [ autoprefixer() ];*

Плагин ***postcss-combine-media-query*** находит в исходниках все медиазапросы с одинаковыми параметрами и склеивает их в один медиазапрос при сборке. Для его подключения нужно:

1. Установить зависимость в проект.

*npm i postcss-combine-media-query --save-dev*

1. В файле ***gulpfile.js*** подключаем плагин.

*const mediaquery = require('postcss-combine-media-query');*

1. Добавляем его в хранилище плагинов.

*const plugins = [ mediaquery() ];*

Плагин ***cssnano*** смотрит убирает форматирование в *css*-файле при сборке проекта для минификации веса. Для его подключения нужно:

1. Установить зависимость в проект.

*npm i cssnano --save-dev*

1. В файле ***gulpfile.js*** подключаем плагин.

*const cssnano = require('cssnano');*

1. Добавляем его в хранилище плагинов.

*const plugins = [ cssnano() ];*

Рассмотрим работу ***PostCSS*** вместе с *Gulp*.

1. Устанавливаем в проекте зависимости для *gulp*-версии *postcss*.

*npm install --save-dev postcss gulp-postcss*

1. Подключаем плагины в файл ***gulpfile.js***

*const postcss = require(‘gulp-postcss’);*

1. В файле ***gulpfile.js*** в функции сборки *css*-бандла указываем используемые плагины.

*const plugins = [ autoprefixer(), mediaquery(), cssnano() ]*

1. В файле ***gulpfile.js*** в функции сборки *css*-бандла добавляем пайп с подключением postcss и его плагинов.

*.pipe(postcss(plugins))*

Для постпроцессинга *html* можно использовать плагин для *Gulp* – ***html-minifier***. Он убирает форматирование в *html*-файле при сборке проекта для минификации веса. Для его подключения нужно:

1. Установить зависимость в проект.

*npm i html-minifier --save-dev*

1. В файле ***gulpfile.js*** подключаем плагин.

*const htmlMinifier = require('html-minifier');*

1. В файле ***gulpfile.js*** добавляем в *html*-функцию переменную с тонкой настройкой минификации и используем её в плагине.

*function html() {*

*const options = {*

*removeComments: true,*

*removeRedundantAttributes: true,*

*removeScriptTypeAttributes: true,*

*removeStyleLinkTypeAttributes: true,*

*sortClassName: true,*

*useShortDoctype: true,*

*collapseWhitespace: true,*

*minifyCSS: true,*

*keepClosingSlash: true*

*};*

*return gulp.src('src/\*\*/\*.html')*

*.pipe(plumber())*

*.on('data', function(file) {*

*const buferFile = Buffer.from(htmlMinify.minify(file.contents.toString(), options))*

*return file.contents = buferFile*

*})*

*.pipe(gulp.dest('dist/'))*

*.pipe(browserSync.reload({stream: true}));*

*}*

**Препроцессинг** – это подход использования новых языком, способных преодолевать ограничения *HTML* и *CSS*.

Основными ограничениями для *HTML* являются монолитность итоговой страницы и сложность создания шаблонов вместо итоговых компонентов. Для решения этих проблем используют специальные программы **шаблонизаторы**. Они позволяют:

* отделить *HTML*-код определённых компонентов от страницы и подключить эти компоненты в нужные места как самостоятельный модуль. Например, одинаковые шапки и подвалы для разных страниц сайта.
* повторно использовать *HTML*-компоненты с разными данными в новом контексте. Например, наполнение карточек товаров в магазине.
* уменьшить повторения в коде или упростить запись, внедряя элементы программирования в *HTML*. Например, массивы и циклы для создания карточек товаров в магазине.

Существует множество популярных шаблонизаторов: [handlebars](https://handlebarsjs.com/), [nunjucks](https://mozilla.github.io/nunjucks/), [liquid](https://shopify.github.io/liquid/). Рассмотрим их работу на примере [pug](https://pugjs.org/api/getting-started.html).

Синтаксис ***Pug*** не использует скобок для обозначения вложенности одних элементов в другие, вместо этого дерево *HTML* описывают через отступы. Атрибуты пишут в круглых скобках, а контент внутри тегов пишется на той же строке, что и имя тега. Поддерживаются массивы, переменные, условия и циклы *each*, *while*.

1. Устанавливаем шаблонизатор в проект.

*npm i gulp-pug --save-dev*

1. В файле ***gulpfile.js*** подключаем плагин.

*const gulpPug = require('gulp-pug');*

1. В файле ***gulpfile.js*** добавляем в *html*-функцию для преобразования файлов на языке *Pug* в *HTML*.

*function pug() {*

*return gulp.src('src/pages/\*\*/\*.pug')*

*.pipe(gulpPug(pretty: true)) // преобразование в HTML без минификации*

*.pipe(gulp.dest('dist/'))*

*.pipe(browserSync.reload({stream: true})); // перезагрузка сервера*

*}*

*exports.pug = pug; // экспортируем функцию*

1. В файле ***gulpfile.js*** добавляем в функцию *watchFiles()* новую задачу:

*gulp.watch(['src/pages\*\*/\*.pug'], pug);*

1. В файле ***gulpfile.js*** обновляем функцию сборки:

*const build = gulp.series(clean, gulp.parallel(pug, css, images));*

1. Теперь при изменении любых файлов шаблонизатора в указанной папке будет производится пересборка проекта и обновление страницы браузера.

Пример *pug*-файла:

*doctype html*

*html(lang="ru")*

*head*

*title Главная страница сайта*

*body.index-page // класс index-page*

*h1.title Привет! // класс title*

*img#main-image(src="/images/pic.png", alt="Описание изображения") // идентификатор main-image*

*p. // перенос строки через точку и вставка элемента в текст через #[]*

*Здесь написан абзац, который #[span.accent может заставить] вас задуматься.*

Для вставки в *pug*-файл шаблонов используется ключевое слово ***extends***. Для обозначения наборов элементов для экспорта/импорта используется ключевое слово ***block***.

Для подключения компонентов в *pug*-файл используется ключевое слово ***include***.

*block content // обозначаем место для импорта шаблона*

*include ../layout/footer // обозначаем место для подключения компонента*

*extends ../layout/default // экспортируем шаблон в основной файл*

*block content*

*h2 Шаблон для экспорта*

Для задания переменных в pug используется ключевое слово ***var***. Переменные также можно вынести в отдельный файл и импортировать через ***block***.

*- var title = ‘Привет переменная’*

*title= title // пример использования*

**Миксины** – это компоненты с уникальным контентом, которые можно переиспользовать много раз. Например:

*mixin card(imgUrl, title, text)*

*article.card*

*.card\_\_image*

*img(src=imgUrl, alt=title)*

*.card\_\_content*

*h2.card\_\_title= title*

*p.card\_\_text= text*

*include ../components/card // пример подключения миксина*

*+card('путь\_до\_изображения', 'Заголовок', 'Текст')*

**Препроцессоры для *CSS*** позволяют сократить код исходников, изолировать код отдельных модулей и добавляют элементы программирования для ускорения написания кода.

Существует не так много препроцессоров для *CSS*: [SASS-SCSS](https://sass-scss.ru/), [LESS](https://lesscss.org/) и [Stylus](https://stylus-lang.com/). На *SASS* код объединяется в блоки с помощью отступов, фигурные скобки не используются. На *SCSS* код очень похож на *CSS*.

Рассмотрим их работу на примере ***SCSS***.