# JavaScript Department
# Style Guide (React)

2020

(RU)

# Конфигурация окружения

Рабочее окружение - это то, чему следует уделить внимание с самых первых минут подготовки к работе. Идеально настроенное окружение - это лучший помощник в работе, который будет экономить огромное количество времени. Если ранее мы всегда давали свободы выбора IDE, то сейчас мы предлагаем Visual Studio Code как практически безальтернативный вариант. Остальные среды интегрированной разработки, как правило, вызывали трудности в тонкой настройке, отнимали много времени и предлагали ограниченное количество функций. Порядок подготовки окружения к работе:

1. Установка Git (с настройкой SSH ключа);
2. Установка Node (как правило, мы используем последнюю четную стабильную версию, но бывает так, что для определенных проектов необходимо загрузить соответствующую версию);
3. Установка Yarn (следует помнить, что выбирая между пакетными менеджерами, следует придерживаться использования одного конкретного, так как разные одновременно подключенные пакетные менеджеры могут начать конфликтовать; даже если на проекте используется npm, следует всегда использовать только его);
4. Установка и настройка Visual Studio Code (перед началом работы, убедиться в том, что проект уже имеет конфигурации ESLint, StyleLint, в проект добавлены зависимости для этих библиотек и консоль вывода ESLint не имеет ошибок);
5. Дополнительные библиотеки и плагины для IDE. Не стоит загружать кучи мусора в IDE, так как избыток различных плагинов может скорее ухудшить ситуацию, чем улучшить. Также крайне не рекомендуется вообще устанавливать какие-либо библиотеки глобально, если в этом нет необходимости.

# Стиль кода

Поддержание код стайла является очень важным. Во-первых, код становится структурным и его легко читать, во-вторых ESLint с правильно написанной конфигурацией позволяет избежать заведомо проигрышных ситуаций, когда структуры написаны некорректно, в-третьих это ускоряет проверку пулл реквестов (если все придерживаются одной конфигурации, иначе только замедляет).

1. Пример конфигурации ESLint;
2. Пример конфигурации StyleLint.

Подключение Prettier позволяет использовать конфигурацию ESLint для автоматического форматирования файла при сохранении или полуавтоматического по требованию (например, по нажатию определенного сочетания клавиш).

# Сборка проекта и зависимости

Стоит уделять особое внимание установке зависимостей. Если зависимость не будет отражена в package.json, приложение не запустится на другой машине, так как необходимая библиотека (или комплекс библиотек, так как зависимости зачастую бывают и вложенные) не будет установлена. В тоже время, не стоит оставлять в зависимостях то, что не будет использовано, так как node_modules постепенно будет наполняться мусором.

Зависимости можно условно разделить на 2 категории: build-time (то, что нужно для сборки приложения) и run-time (то, что нужно для работы приложения). Все build-time зависимости мы устанавливаем в devDependencies, все run-time, соответственно, в dependencies.

Для того, чтобы убедиться, что приложение запустится на другой машине, следует вручную удалить директорию node_modules (предварительно убедившись, что не установлены глобальные зависимости, которые могут “подцепиться” вместо “потерянных” локальных) и вызвать две команды: install и start.

# Общие рекомендации к написанию кода

Данные правила следует использовать во время написания кода, а не для исправления, так как любое исправление - это дополнительное время.

## Git

1. Всегда следуй Git-flow *(описан в памятке и в пункте “Порядок работы с Git репозиторием”)*;
2. При появлении комментариев к своему pull-request обязательно отреагируй на него: любой вопрос стоит сначала обсудить, а уже потом начинать работу над исправлениями. После исправления обязательно добавь комментарий “Fixed”.

## Общие правила

1. Статический текст и “магические” *(примеры ниже в пункте “Магические числа”)* числа должны быть занесены в константы;
2. При использовании одного и того же значения используем константы;
3. Минификацией и углификацией занимается не разработчик, а webpack;
4. Всегда помни о том, что с твоим кодом кто-то будет работать. При реализации используй имена, которые будут понятны всем. Код должен открыть человек, который его никогда не видел и понять, что в нем происходит: по именам переменных, порядку операций и т.д;
5. Комментарии в коде должны нести смысл, а не быть каким-то украшением. Мы не занимаемся тюнингом кода. Если все же есть намерение написать комментарий, который будет объяснять, как что-то работает - задумайся, может стоит переписать код, раз его нужно объяснять.

6. Всегда используй path.alias для импорта кастомного кода *(описано ниже в пункте “Алиасы (path aliases)”)*;
7. Соблюдай все правила для отступов, пробелов, переносов, которые тебе предлагает ESLint. Все проблемы, связанные с код стайлом должны быть исправлены. Инструкция “eslint-disable” не является исправлением. Не забывай, что в сложных ситуациях всегда можно обратиться к коллеге за помощью.

## Компоненты и контейнеры

1. Соблюдай структуру директории компонента (описано в пункте “Структура компонента”);
2. Допускай вложенность компонентов только в том случае, если вложенные компоненты никогда не будут использоваться извне родительских. Всегда придерживайся одного подхода и стиля;
3. Обязательно описывай propTypes и defaultProps. Каждое свойство должно быть детально описано. Если видишь, что один и тот же propType встречается 2 и более раз - выноси его в отдельную директорию и делай импорт вместо того, чтобы создавать копии;
4. Если компонент не использует методы жизненного цикла - это функция;
5. Не завязывай компоненты на внутренние данные. Все максимально должно быть настраиваемо из props;
6. Не определяй функции (в том числе и стрелочные) в качестве props. Мы передаем только ссылки на эти функции. Помни про useCallback и useMemo;
7. Значения, получаемые из store и описываемые в mapStateToProps работают через библиотеку reselect.

## Стили

1. Размеры текста задаются через rem;
2. Все значения, которые существуют в темах задаются из темы;
3. Соблюдай код стайл и в стилях: отступы, пробелы и т.д;
4. Если ты описал какой-то стиль 2 раза (не важно, в разных файлах или нет) или более - это отдельный компонент или дополнение к существующему, похожему на то что реализовывал;
5. При добавлении каких-либо значений в тему, это нужно сделать и в других темах (если реализованы несколько тем), не только в той, с которой сейчас работаешь.

## Именование переменных и функций

1. Имя должно отражать суть того, что происходит внутри:
    a. если это переменная с данными - то что в ней хранится;
    b. если это функция - обязательно наличие глагола, который описывает действие внутри.
2. Именование с большой буквы только для компонентов и классов;

3. Даже если есть понимание того, что переменная может быть временной , следует ее называть так, чтобы если она остается в кода не было названий string1 и подобного;
4. Необходимо соблюдать грамматику при именовании переменных.

### Магические числа

Это один из подвидов статического кода, но только с числами. Например, если обратить внимание на следующий код, то в лучшем случае придется задуматься, что это значит, а в худшем - действовать вслепую:
`const t = 86400` // никто не знает, что такое “t”, и что такое 86400.

`const SECONDS_IN_DAY = 86400` // определяем константу с читаемым именем
`const broadcastTimeout = SECONDS_IN_DAY` // именуем переменную читаемым именем;

### Алиасы (path alias)

Path alias настраиваются в конфигурации webpack. При использовании таких инструментов, как “create-react-app” или подобных, конфигурацию приложения следует расширить для того, чтобы получить возможность внести изменения в базовый файл конфигурации webpack, например при помощи “react-app-rewired” или подобных.

# Обработка ошибок

Можно сказать, что неправильная обработка ошибок - это большая “головная боль” начинающего разработчика. Под обработкой ошибок, простыми словами, подразумевается случай, когда что-то пошло не так. Без разницы: не получилось преобразовать данные, отсутствует сеть, сервис недоступен, или просто “вылетела” TypeError - это все ошибки и критические ситуации, которые важно правильно уметь обрабатывать (не вдаваясь в тонкости бизнес-логики какого-либо конкретного приложения). Поэтому, помимо различных проверок, следует пользоваться конструкциями try-catch-finally и Promise-then-catch-finally.

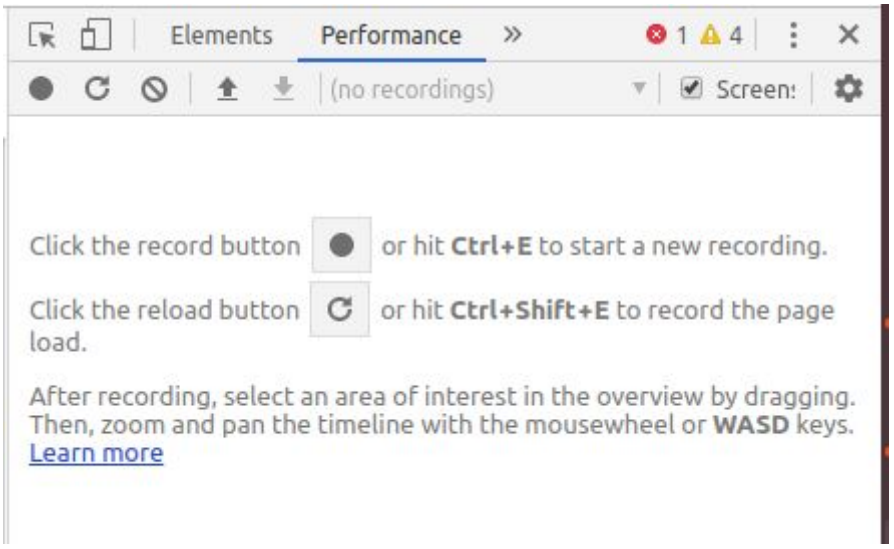
Следует пользоваться единым стилем обработки ошибок во всем приложении. Один раз написанный обработчик заставит разработчика практически забыть о проблеме обработки ошибок. Для запросов и сервисов это особенно важно, когда есть единый стандарт обработки. Самый наилучший вариант - вынести обработчик в библиотеку или стартер.

Не следует “проглатывать” или “хардкодить” текста ошибок. Тексты ошибок - это не бизнес-логика. Это вещь, которая может быстро и часто меняться и однозначно указывать на проблему, которая спровоцировала ошибку. В случаях, когда необходимо оповестить пользователя, сначала следует определить проблему программно, а после этого сообщить ему дальнейшие действия, не забыв залогировать реальное сообщение.

# Отладка кода

Не следует понимать множество вызовов “console.log” за отладку кода. В сложных ситуациях или в местах, где следует наблюдать за различными данными, можно использовать ключевое слово “debugger” и расставить точки останова. Это позволить шаг за шагом продолжать выполнение приложения и наблюдать за тем, что там происходит. Тем не менее, даже при использовании логов в консоль, не стоит забывать о том, что в конечном продукте не должно быть никаких отладочных вызовов этого метода (то есть, если “console.log” используется для отладки, то сразу после этого он должен быть убран).

Помимо debugger, можно использовать “profiler”, вшитый в большинство браузеров для оценки количества запусков функций, объема памяти, который они занимают, времени вычислений, которые они тратят и многое другое, что позволит выявить реальную проблему и произвести оптимизации максимально качественно и целостно.




Разрабатывая приложение, основанное на React, не следует забывать о дополнительных инструментах, таких как React DevTools, Redux DevTools. Эти расширения, как правило, устанавливаются в браузер и позволяют продемонстрировать компонентную модель и состояние всех узлов, дать доступ к хранилищу приложения и его диспетчеру.

# Структура и общая архитектура проекта

У большинства разработчиков есть уже свой сформировавшийся набор инструментов и готовых решений для старта проекта (boilerplates, starter-kits и прочее). Иногда, в зависимости от технологического стека, сборка базового приложения (скелета приложения) производится с нуля. Чаще всего в таких случаях за основу берется create-react-app или одна из его вариаций с уже включенными общими библиотеками.

Различные проекты могут иметь различный набор библиотек и практик, что может косвенно оказать влияние на конечную структуру проекта, но не смотря на это возможно выделить общую (подробное описание по каждому пункту будет приведено ниже):

- *build **
- public
- src
- .babelrc
- *.env **
- .eslintrc.json
- .gitignore
- .stylelinntrc
- README.md
- config-overrides.js
- package.json
- yarn.lock

## build

Эта директория создается автоматически (в процессе работы команды “build”), ее не нужно создавать вручную или подготавливать. Как правило, она указывается в конфигурации webpack как директория, которая хранит сборку приложения и является готовым артефактом для загрузки на сервер. Важно помнить, что в процессе разработки она не участвует никак. Активная разработка запускается посредствам команды “start”, что несколько отличается от обычной сборки, так как в данном случае сборка происходит в in-memory-file-system.

## public

Директория хранит статику (индексную страницу, манифест приложения, иконки, robots.txt и прочее).

## src

Корневая директория, хранящая исходный код приложения. О ней будет более подробно рассказано ниже.

## .babelrc

Хранит в себе конфигурацию babel. Какие пресеты должны использоваться на проекте, плагины, различия в конфигурации относительно окружения под которым запущен проект (разработка, продакшн, тестирование и прочее). Как правило, зачастую не требует доработок, если сгенерирован при помощи create-react-app или прочих утилит.

## .env

Файл хранит в себе переменные окружения, в том числе и секретные ключи. Важно помнить, что в таком виде он не должен попадать в репозиторий и обязательно должен быть помещен в .gitignore. Для того, чтобы указать, что может храниться в .env файле и какое у переменных назначение, создается .env-example, где описываются переменные и в качестве комментария указывается что в этих переменных может храниться. Перед началом разработки этот файл следует скопировать и переименовать в .env, дополнительно убедившись, что он занесен в блэк-лист гита. Вместе с этим, возможно создание нескольких копий .env для указания различных переменных относительно окружения, под которым запускается проект: .env .env.production .env.qa и прочие.

## eslintrc.json

Данному файлу следует уделить большую долю внимания, так как именно он и хранит в себе все правила код стайла. В нем можно задать базовую конфигурацию для наследования, различные плагины и переопределения правил базовой конфигурации. Как правило, мы используем конфигурации “standard” и “standard-react”. В качестве примера могу привести конфигурацию своего личного starter-kit: https://github.com/BLRplex/ig-react-starter/blob/master/.eslintrc.json

## .gitignore

Файл хранит в себе пути, которые не должны попасть в репозиторий. Как правило, это node_modules, директория с отчетом о покрытии тестами, сборка проекта, .env файлы (зачастую уже определены и различные вариации относительно окружения), логи, настройки IDE (если не предполагается, что вся команда должна работать под одной конкретной IDE с конкретным набором настроек) и прочее.

## .stylelintrc

Назначение файла схоже с .eslintrc.json, но только применимо к стилям. Современные проекты зачастую содержат Styled-Components, что по сути является CSS-in-JS и данная библиотека как раз поддерживает даже такое.

### README.md

Один из самых важных файлов проекта. Он содержит подробное описание проекта, конфигураций и всего, что необходимо инженеру для максимально быстрой установки и запуска приложения. Чем более подробно он описан, чем меньше в нем воды и мусора, тем проще получить основную информацию и начать работу. В этом же файле можно описать все переменные окружения, правила и код стайл проекта. Говоря другими словами, это отправная точка для любого разработчика, начавшего работу на проекте.

### config-overrides.js

Актуален только для create-react-app, когда есть необходимость вмешаться в webpack.config.js, который скрыт от разработчика. Для решения этой проблемы существует библиотека create-app-rewired, которая как раз и использует эту конфигурацию. При самостоятельном подключении, следует вручную заменить вызовы “react-scripts” в файле “package.json” на “create-app-rewired”.

### package.json

Хранит в себе основную информацию о проекте, зависимостях, команды, скрипты и прочее.

## Структура src

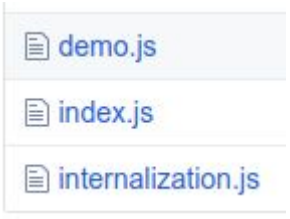
Так как на различных проектах может использоваться различный набор библиотек, универсально описать структуру этой директории не возможно. В данном примере рассматривается структура с использованием следующего стека: React, Redux, Redux-Saga, Styled-Components, Jest, React-Intl, Prop-Types, Formik.

- actions
- components
- constants
- *e2e **
- internalization
- prop-types
- reducer
- sagas
- theme
- *App.jsx **
- *Router.jsx **
- index.js
- *setupTests.js **
- store.js

## actions

Директория содержит Redux Actions. Это могут быть как базовые объекты, так и функции Action creators. Рекомендуется дополнительно создавать файл index.js, который будет производить экспорт всего содержимого всех файлов с экшенами:

actions/




actions/index.js

```
2 lines (2 sloc)  57 Bytes
1  export * from './demo'
2  export * from './internalization'
```

actions/internalization.js

```
1  import { SET_LOCALE } from '@/constants'
2
3  export const setLocale = locale => ({
4    payload: locale,
5    type: SET_LOCALE,
6  })
```

## components

Все компоненты приложения расположены в этой директории. Система разделения кода также может различаться и в разных проектах иметь другой вид.

- blocks - содержит блоки, такие как Header, Footer, Navigation Menu, F.A.Q. Может разделяться на глобальные и уникальные блоки, которые используются только единожды (например, Header) и локальные (например UsersList, Avatar, Pagination)
- controls (fields) - содержит элементы управления, такие как кнопки, поля ввода, поля выбора элементов и прочее
- forms - содержит формы, в данном случае, интегрированные с Formik
- layouts - хранит разметки приложения (например: стандартная, в которой есть хедер, футер и контент; упрощенная, в которой хедер и футер отсутствуют или наоборот, расширенная с дополнительными элементами для определенных страниц)
- pages - страницы приложения, можно хранить как линейно, так и иерархически
- wrappers - обертки над компонентами или компоненты высшего порядка

В зависимости от технологического стека проекта или “вкусовых предпочтений” инженера, некоторые директории могут объединяться с другими или дополнительно разделяться иным способом.

Ниже будет детально описана структура самого компонента и его файлов.

## constants

В этой директории хранятся константы приложения (не путать с переводами и секретными ключами, которым не место здесь). Как и в случае с actions, рекомендуется создавать дополнительно index.js, который будет производить множественный экспорт всех переменных. В этой директории может быть тоже своя структура и разделение кода, например таким образом:

- actions.js - хранит типы экшенов

```
export const DEMO_ACTION = 'DEMO_ACTION'

export const SET_LOCALE = 'SET_LOCALE'
```

- endpoints.js - хранит пути к внешним сервисам

```
export const DEMO_ENDPOINT = 'https://example.com'
```

- forms.js - хранит названия форм

```
export const DEMO_FORM = 'demoForm'
```

- fields.js - хранит названия полей

```
export const FIRST_NAME_FIELD = 'firstName'
export const LAST_NAME_FIELD = 'lastName'
```

- index.js - экспортирует всех остальные файлы

```
export * from './actions'
export * from './paths'
export * from './endpoints'
export * from './forms'
export * from './internalization'
```

- internalization.js - хранит общую информацию по интернализации

```
export const ENGLISH = 'en'
export const RUSSIAN = 'ru'
```

- paths.js - хранит локальные пути роутера клиентского приложения

```
export const LANDING_PAGE_PATH = '/'
```

## e2e

Это опциональная директория и с большой вероятностью ее может не быть. На проекте, который приведен в примерах, интегрировано end-to-end тестирование со скриншотами. В директории e2e находятся основные настройки, в том числе и разрешения экранов, для которых будет производиться тестирование.

## internalization

Директория может отсутствовать в проектах, в которых не реализована мультиязычность или располагаться в другом месте, так как это может зависеть от целевой библиотеки

## prop-types

Мы всегда определяем проп-таймы компонентов. Иногда бывает так, что два и более компонента используют общие проптайпы или сущности имеют интерфейсы. В таких случаях, следует помещать общие проптайпы в отдельную директорию и делать импорт вместо копипаст. Например, если компонент принимает в качестве свойства строку или компонент, можно создать проп-тайп следующим образом и во всех подобных ситуациях импортировать его:

```
import pt from 'prop-types'

export const childrenPropType = pt.oneOfType([
  pt.string.isRequired,
  pt.element.isRequired,
])
```

## reducer

Корневой редьюсер приложения, обязательно должен иметь index.js, который загружает все дочерние редьюсеры. Все описанное в этой категории может быть не актуально, если приложение использует модульную архитектуру (тоже самое касается и actions).

Пример корневого редьюсера:

```
import { combineReducers } from 'redux'

import internalization from './internalization'

export default combineReducers({
  internalization,
})
```

Пример отдельно взятого редьюсера:

```
import { SET_LOCALE, ENGLISH } from '@/constants'

const initialState = {
  active: ENGLISH,
}

export default function (state = initialState, action) {
  switch (action.type) {
    case SET_LOCALE: return {
      ...state,
      active: action.payload,
    }

    default: return state
  }
}
```

## sagas

Может отсутствовать в некоторых проектах. Redux-saga - это библиотека, которая призвана упростить и улучшить побочные эффекты (т.е. такие действия, как асинхронные операции, например, загрузки данных, и "грязные" действия, такие, как доступ к браузерному кешу), сделать лёгкими в тестировании и лучше справляться с ошибками. Не стоит забывать, что и здесь тоже следует хорошо организовать структуру и разделение кода. Саги можно “объединять” в логические структуры вызовом fork.

## theme

Создание темы и стайлгайда - это признак хорошего тона. Вместо ручного указания цветов, отступов, размеров и названий лучше использовать константы, которые следует импортировать. Это значительно упростит внесение изменений в стили в будущем и создаст однозначность в стилистике приложения, не допустив того, что разные компоненты отличаются друг от друга стилистически.

Здесь же можно указать и глобальные стили приложения (html, body и прочее, что находится вне корневого элемента приложения, если в этом есть необходимость).

## App.jsx

Этот файл может носить и другое имя. Здесь он не переименован по причине того, что так же он называется и в create-react-app. То есть, для того, чтобы можно было связать файл и базового набора и собственной реализации. Является корневым компонентом. Здесь следует определить глобальные компоненты, которые относятся ко всему приложению вне зависимости от страницы, которая открыта. Например:

```
import React from 'react'
import { Provider } from 'react-redux'

import Router from '@/Router'
import { getStore } from '@/store'
import ThemeProviderWrapper from '@/components/wrappers/ThemeProvider'
import Internalization from '@/components/wrappers/Internalization'

function App () {
  return (
    <Provider store={getStore()}>
      <Internalization>
        <ThemeProviderWrapper>
          <Router />
        </ThemeProviderWrapper>
      </Internalization>
    </Provider>
  )
}

export default App
```

Provider подключает Redux, Internalization является упрощенной оберткой над IntlProvider от react-intl, ThemeProviderWrapper позволяет применить глобальные стили и подключить Theme-Provider от Styled-Components, Router является глобальным роутером приложения.

## Router.jsx

Глобальный роутер приложения. Возвращает маршруты, которые должны быть доступны вне зависимости от открытой страницы, например:

```
<Router>
  <React.Suspense fallback={<Loader />}>
    <Switch>
      <Route
        exact
        path={LANDING_PAGE_PATH}
        component={LandingPage} />

      <SecuredRoute
        path="/success"
        component={LandingPage}
        strategy={successStrategyExample} />

      <SecuredRoute
        path="/failure"
        component={LandingPage}
        strategy={failureStrategyExample} />

      <Route path="*" component={NotFoundPage} />
    </Switch>
  </React.Suspense>
</Router>
```

Во многих реализациях может отсутствовать React.Suspense и SecuredRoute. В примере реализована динамическая подгрузка роутера, что может быть актуально только для крупных приложений с целью уменьшения времени загрузки и увеличения быстродействия:

```
const LandingPage = React.lazy(() => import('@/components/pages/Landing'))
const NotFoundPage = React.lazy(() => import('@/components/pages/NotFound'))
```

SecuredRoute является компонентом высшего порядка, который определяет, есть ли у клиента права доступа к определенному маршруту (в примере используется стратегия для определения прав: в каких-то маршрутах достаточно проверить, что пользователь авторизован, а в других убедиться, что у него есть соответствующая роль. В случае успеха возвращается необходимый маршрут, в других случаях пользователю показывается уведомление о том, что у него нет прав на посещение указанной страницы или происходит перенаправление на другую публичную страницу.

### index.js

По сути является точкой входа в приложение, единственная его обязанность заключается в монтировании корневого компонента в DOM:

```
import React from 'react'
import ReactDOM from 'react-dom'

import App from '@/App'

ReactDOM.render(<App />, document.getElementById('root'))
```

### setupTests.js

Аналогично, опциональный файл, которого может не быть или может быть именован иначе. Его назначение заключается в инициализации тестового окружения.

### store.js

Формирует глобальное хранилище приложения. Здесь можно дополнительно подключить middlewares. Также, в зависимости от окружения, подключать дополнительные middlewares, например dev-tools только для development окружения. Здесь же можно реализовать синглтон для стора:

```
export const getStore = () => {
  if (!store) {
    store = process.env.NODE_ENV === 'development'
      ? createDevelopmentStore()
      : createProductionStore()

    sagaMiddleware.run(rootSaga)
  }

  return store
}
```

## Структура компонента

Следует всегда придерживаться одной и той же стилистики, в том числе и в компонентной модели, и в структуре самого компонента. Мы предлагаем следующую модель:

- component.jsx - хранит в себе компонент и ничего более;
- container.js (опционален) - хранит mapStateToProps, mapDispatchToProps и connect. Также, если используется recompose или подобные библиотеки, функциональность, связанную с ними стоит также описывать в этом файле;

- index.js - при наличии контейнера экспортирует по умолчанию контейнер, в противном случае сам компонент. Это сделано для упрощения импорта и лишения разработчика раздумий о том что в данной ситуации нужно импортировать: компонент или контейнер. В случаях, когда нужен только компонент - производится прямой импорт компонента минуя индекс, во всех остальных случаях используем “автоматическую” логику;
- component.test.jsx (опционален) - может существовать только при подключенных тестах. Если приложение покрывается тестами, следует покрывать тестами все возможное (естественно, если в этом есть необходимость);
- styles.js - хранит стили Styled-Components, импортируется непосредственно самим компонентом.

## Алиасы и порядок импортов

Как уже говорилось выше, следует использовать всегда один стиль. Это также касается импортов. Зачастую можно встретить импорты с относительными путями, которые выглядят как “../../../../components/Header”, что однозначно будет вызывать затруднения с поиском места расположения компонента и миграции. Причем, чем приложение больше и сложнее, тем больше масштабы проблемы. Для того, чтобы это избежать используют алиасы (в примерах кода можно видеть символ “@” в импортах). Алиас - это сокращенное имя абсолютного пути, зачастую его связывают с директорией “src”.

При использовании create-react-app их можно с легкостью настроить подключив библиотеку “react-app-rewired” и описав конфигурацию в файле “config-overrides.js”. Пример файла для конфигурации алиаса “@”:

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

module.exports = function (config, env) {
  config.resolve = {
    ...config.resolve,
    alias: {
      ...config.resolve.alias,
      '@': path.resolve(__dirname, './src'),
    },
  }

  return config
}
```

Можно заметить, что здесь вместо привычных импортов используется “require”, что с одной стороны может показаться игнорированием требования делать все в одном стиле. Но, на самом деле это не так. React приложение попадает непосредственно в Babel, а данная конфигурация переопределяет конфигурацию webpack задолго до запуска Babel. Поэтому, здесь придерживаемся правил для Node.

# Манипуляции над данным

Всегда придерживайся оптимальных алгоритмов для манипуляции над данными. Да, многое можно реализовать при помощи циклов, сделав код медленнее и менее очевидным, но исходя из пункта “Общие рекомендации к написанию кода” об общих рекомендациях, привыкаем делать сразу правильно (“_” ссылается на библиотеку lodash).

- Копия объекта (массива) - Object.prototype.assign, Array.prototype.concat, Array.prototype.slice, Spread (внимание, все эти способы создают поверхностную копию объекта, это значит все все ссылки внутри него тоже будут скопированы), _.clone, _.cloneDeep (реализуют алгоритм структурированного клонирования, имеет преимущества перед OBJ->JSON->OBJ);
- Сравнения - Все примитивы сравниваются по значению (не забываем про особенности NaN и Infinity), а объекты по ссылке; при необходимости сравнить объекты (массивы) по содержимому, на помощь придется _.isEqual;
- Поиск элемента:
    - Array.prototype.find - возвращает искомый элемент используя коллбэк функцию
    - Array.prototype.findIndex - возвращает индекс искомого элемента, используя коллбэк функцию
    - Array.prototype.includes - возвращает логическое значение, если массив содержит искомый элемент
    - Array.prototype.indexOf - возвращает индекс искомого элемента
    - Array.prototype.lastIndexOf - возвращает последний индекс искомого элемента
    - Array.prototype.some - возвращает логическое значение, используя коллбэк функцию

Крайне рекомендуется знать все методы массивов и объектов, а также общие методы общих вспомогательных библиотек (lodash, underscore).

# Верстка и тема

Сейчас мы все чаще используем технологию CSS-in-JS, что позволяет создавать изолированные и экспортируемые стили, а точнее компоненты высшего порядка. Это позволяет интегрировать стили непосредственно в компонентную модель приложения, причем уровень изоляции не допускает случайного переопределения существующих.

Для реализации этой технологии мы пользуемся библиотекой Styled Components. Она также просто интегрируется с CSS фреймворками и компонентными библиотеками, такими как AntDesign, Material-UI (Materialize), Bulma, Bootstrap, Foundation, Semantic UI и прочими. Более того, Styled Components позволяют определить глобальные стили (для DOM элементов вне родительского элемента приложения) и тему.

Темизация - это отличный инструмент, который позволяет определить цвета, отступы, шрифты и прочее, а потом их использовать в качестве констант. Это делает внешний вид приложения единым и позволяет изменять цвета просто и быстро. Все что нужно - это в одном месте один раз отредактировать одну константу. Более, таким образом можно определить различные конфигурации темы и переключать их “на лету”.

# Тестирование

Покрытие приложения тестами значительно улучшает стабильность приложения. Существует три типа тестов: unit-тесты, интеграционные тесты и снапшот (скриншот) тесты.

## Unit - тесты

Unit-тестирование, или модульное тестирование - это тестирование, которое позволяет проверить корректность работы методов и классов приложения. Правильно написанные unit-тесты должны обладать следующими свойствами:

- быть достоверными;
- не зависеть от окружения, на котором они выполняются;
- легко поддерживаться;
- легко читаться и быть простыми для понимания (даже новый разработчик должен понять что именно тестируется);
- соблюдать единую конвенцию именования;
- запускаться регулярно в автоматическом режиме.

Нужно тестировать тот код, который имеет зависимости. При этом, если код метода проекта слишком сложный, нужно сначала провести рефакторинг узкого участка, а уже потом писать тесты. Каждый тест должен проверять только одну вещь. Требования, касающиеся написания исходников программы, касаются и тестов. Код должен быть чистым, аккуратным, логически выверенным. Качественно написанный unit-тест избавит в будущем от множества проблем с поддержкой программного продукта.

## Интеграционные тесты

При интеграционном тестировании, мы проверяем работу сервисов и компонентов при их взаимодействии между собой. Отличие от unit-тестов в том, что при интеграционном тестировании тест:

- использует базу данных;
- использует сеть для вызова другого приложения;
- использует внешнюю систему (например, очередь или почтовый сервер);
- полагается не на исходный код, а на сборку приложения.

С помощью интеграционных тестов можно обнаружить проблемы интеграции компонентов, проблемы с транзакциями, триггерами и процедурами БД, неправильные контракты с другими модулями, API и сервисами, проблемы безопасности и производительности сервиса, взаимные блокировки и прочее.

### Снапшот (скриншот) тесты

Позволяют предотвратить визуальную регрессию приложения и его компонентов до того, как обновление будет добавлено в общую кодовую базу. Позволяют одновременно тестировать на различных размерах дисплеев (от телефонов до больших дисплеев с Retina).

# Порядок работы с Git репозиторием

Поддержание порядка в процессах работы с репозиторием также является важной частью разработки. Это позволяет быстро применять или отменять новую функциональность, иметь контроль над версионированием и распределением работы в команде между ее участниками. Не стоит добавлять в репозиторий закомментированный код или код, необходимый только для отладки. Считаем так, если код закомментирован - значит он не нужен. Если код не нужен - значит ему нечего делать в репозитории. Всякие временные фичи или экспериментальное поведение можно реализовывать в соответствующих ветках и при необходимости мержить их в целевую.

Мы предлагаем следующий порядок:

1. Перед тем как приступить к выполнению задачи, следует обновить основную ветку и создать новую, руководствуясь следующим:
    - имя ветки должно состоять из двух частей: тип задачи и краткое описание;
    - тип задачи может быть “feature” (реализация новой функциональности) и “defect” (исправление существующей функциональности);
    - краткое описание должно состоять из 2-7 слов, написанных в нижнем регистре и разделенных знаком “-”. Например, если задача была в реализации “подвала” приложения, то ее краткое описание будет “implement-footer”, а полное название ветки - “feature/implement-footer”.
2. После реализации функциональности или исправления результат следует обязательно протестировать, в том числе и в других браузерах, так как в них что-то может работать иначе;
3. Не следует делать больше одного коммита без видимой на то причины;
4. Запрос на слияние (merge request, pull request) создаются после загрузки ветки в репозиторий. Перед тем, как отправить запрос на слияние, обязательно нужно контрольно проверить его самостоятельно и при необходимости исправить найденные проблемы;
5. После этого следует отправить ссылку техническому руководителю проекта и приступить к решению новой задачи;
6. При наличии вопросов в запросе на слияние или необходимости исправить что-либо, следует переключиться на необходимую ветку, внести изменения, загрузить их в репозиторий и оповестить руководителя.