РАЗРАБОТКА ВЕБ-САЙТОВ*, TYPESCRIPT*, REACTJS*, JAVASCRIPT*

Окружение для разработки веб-приложений на TypeScript и React: от 'hello world' до современного SPA. Часть 1

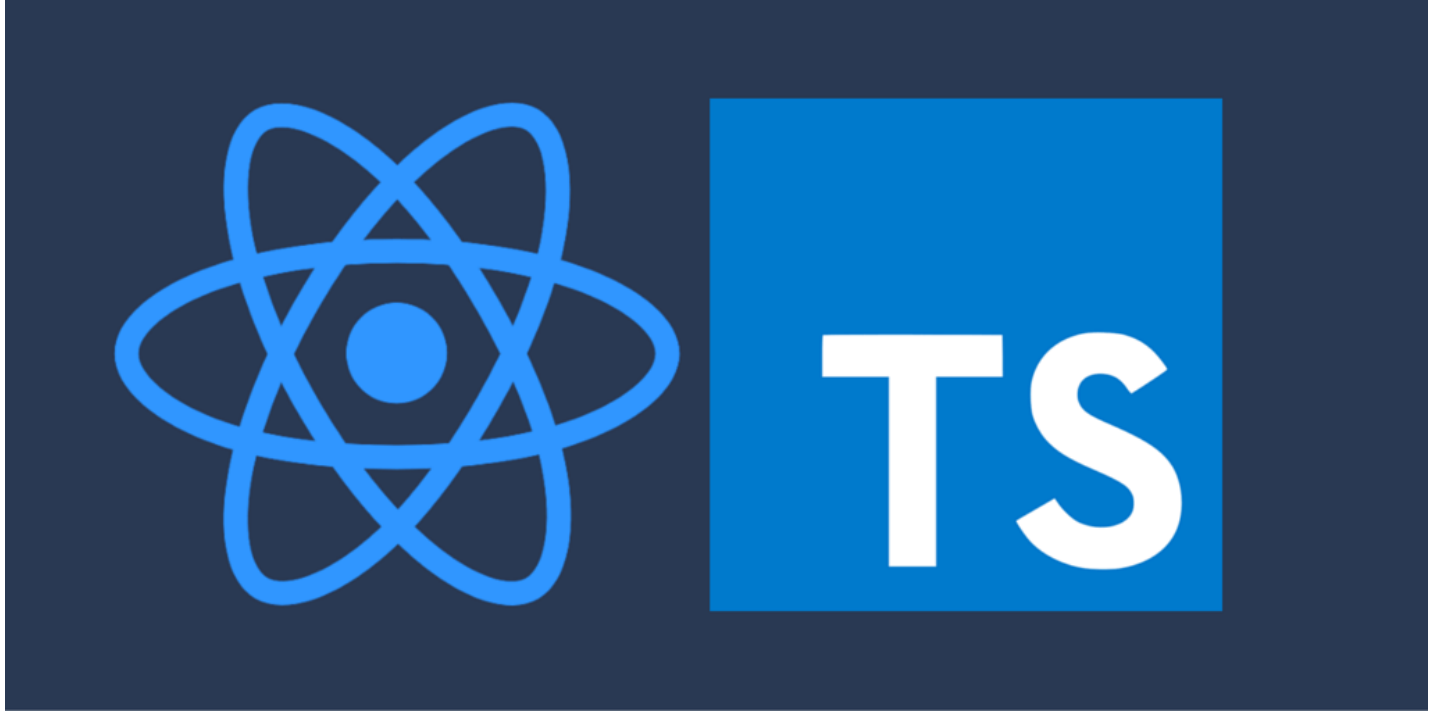
ИЗ ПЕСОЧНИЦЫ

SuperOleg39ru 21 октября 2017 в 21:35 18,3k

Цель данной статьи — вместе с читателем написать окружение для разработки современных веб-приложений, последовательно добавляя и настраивая необходимые инструменты и библиотеки. По аналогии с многочисленными starter-kit / boilerplate репозиториями, но наш, собственный.

Так же, автор пишет эту статью для структурирования собственных мыслей, знаний и практического опыта, и получает хорошую мотивацию на изучение новых аспектов разработки.

Автор полностью открыт для доработки и исправления текущей статьи, и надеется на превращение итогового материала в актуальный и удобный справочник, интересный и для профессионалов, и для желающих опробовать новые для них технологии.



Статья не рассматривает подробный синтаксис TypeScript и основы работы с React, если читатель не имеет опыта использования указанных выше технологий, рекомендуется разделить их изучение.

Ссылка на вторую часть статьи

Немного об используемых технологиях:

Написание проекта на TypeScript влечет за собой множество трудностей, особенно при первом знакомстве с языком. На взгляд автора, преимущества строгой типизации стоят этих усилий.

Помимо возможностей самого языка, компилятор TypeScript генерирует JavaScript код под все версии стандарта, и позволяет отказаться от использования Babel в проекте (автор не имеет ничего против этого замечательного инструмента, но одновременное использование TS и Babel вносит небольшую

путаницу на старте).

React — зарекомендовавшая себя библиотека для создания веб-интерфейсов, с огромным сообществом и инфраструктурой.

Недавно вышла новая версия библиотеки со множеством улучшений и переработанной документацией.

Для сборки проекта мы будем использовать Webpack — лучший друг frontend разработчика. Базовые настройки этого инструмента очень просты в изучении и использовании. Серьезно.

Используемые версии инструментов и библиотек

Начнем!
Репозиторий проекта содержит код в отдельных ветках под каждый шаг.

Шаг первый — добавление TypeScript в проект.

Для просмотра итогового кода:

```
git checkout step-1
```

Установка зависимостей:

```
npm i webpack typescript awesome-typescript-loader --save-dev
```

awesome-typescript-loader — TypeScript загрузчик для webpack,

считается быстрее основного конкурента — ts-loader.

Для исходников нашего проекта создадим папку `src`.
Результаты сборки будем отправлять в `dist`.

Базовые настройки для компилятора TypeScript — файл `tsconfig.json` в корневой директории проекта

tsconfig.json

Базовые настройки сборщика — файл `webpack.config.js` в корневой директории проекта:

webpack.config.js

Внутри `src` создадим файл `index.ts` с любым кодом, использующим синтаксис TypeScript, например:

index.ts

Команда для компиляции и сборки нашего кода:
`webpack` — разовый билд проекта

В итоговом файле `dist/app.bundle.js` внутри webpack модулей вы увидите аккуратный и читаемый JavaScript код выбранной нами версии стандарта.

Созданное нами окружение легко расширить любыми библиотеками, и удобно использовать для создания прототипов

(Ваша Любимая Технология + TypeScript).

Идем дальше!

Шаг второй — создание крохотного React приложения.

Для просмотра итогового кода:

```
git checkout step-2
```

Установка зависимостей:

```
npm i webpack react react-dom --save
npm i webpack @types/react @types/react-dom html-
webpack-plugin clean-webpack-plugin --save-dev
```

html-webpack-plugin — плагин для генерации html-файла с подключенными результатами сборки.
clean-webpack-plugin — для очистки директории с результатами сборки.
@types/react и **@types/react-dom** — пакеты с декларацией соответствующих JS библиотек, дающие компилятору TS информацию о типах всех экспортируемых модулей.

Большая часть популярных JS библиотек имеет декларации, иногда они находятся в исходных файлах проекта, иногда — в замечательном репозитории DefinitelyTyped, который активно развивается благодаря сообществу, и в случае отсутствия или ошибок в существующей декларации, вы можете с легкостью

вносить свой вклад для исправления этих проблем.

Внутри `src` создадим файл `index.html` с элементов для монтирования корневого react компонента:

index.html

Обновляем настройки webpack:

webpack.config.js

Обновим настройки компилятора TypeScript:

tsconfig.json

Перейдем к компонентам.

Необходимо изменить расширение у `index.ts` на `index.tsx`. Напишем код нашего компонента, и отобразим его на странице:

index.tsx

Добавим команду для компиляции и сборки нашего кода:

`webpack-dev-server` — поднимаем сервер с нашим приложением, страница index.html будет доступна по адресу — `http://localhost:8080/`.
Так же, webpack будет осуществлять автоматическую пересборку проекта, при изменении исходных файлов.

На данном этапе могут возникнуть вопросы к размеру сборки — автор уделит особое внимание разделению на production и development сборки, в следующих шагах. На первых этапах стоит акцент на минимально необходимых настройках и библиотеках, для полного осознания процесса.

Шаг третий — рецепты приготовления React и TypeScript

Для просмотра итогового кода:

```
git checkout step-3
```

Зависимости на этом шаге не меняются.
Рекомендуется ознакомиться с обобщениями на этом этапе — generics (дженерики)

Более подробно о стандартных React паттернах можно узнать из этой статьи.

1) **Стандартный компонент, имеющий свойства и состояние**
Создадим компонент `simple.tsx`, который будет выводит контролируемое поле ввода:

simple.tsx

2) **Компонент высшего порядка**
Описание компонентов высшего порядка в официальной документации React — по ссылке

Статья, подробно расписывающая написание компонента высшего порядка на TypeScript (примеры из этой статьи частично заимствованы автором) — по ссылке

Если коротко, компонент высшего порядка (далее hoc) — это функция, которая принимает аргументом компонент (и по желанию дополнительные опции), и возвращает новый компонент, который выводит старый в методе `render`, передавая ему свои свойства и состояние.

Сигнатура выглядит так: `(Component) => WrapComponent => Component`

Так как TypeScript строго следит за тем, какие свойства мы передаем в компоненты, нам нужно определиться с интерфейсами этих свойств.
OriginProps — уникальные свойства компонента, hoc ничего о них не знает, только передает в компонент.
ExternalProps — уникальные свойства hoc.
InjectedProps — свойства, которые мы будем передавать в компонент из hoc, рассчитываются на основе ExternalProps и State.
State — интерфейс состояния hoc. Так как мы будем передавать компоненту все состояние hoc, State не может иметь свойств, которые отличаются от InjectedProps (либо мы должны передавать доступные свойства, не используя оператор расширения).

Перейдем к коду, напишем простой счетчик нажатий кнопки. Создадим папку `hoc`, в ней компонент `displayCount.tsx` и hoc

withCount.tsx

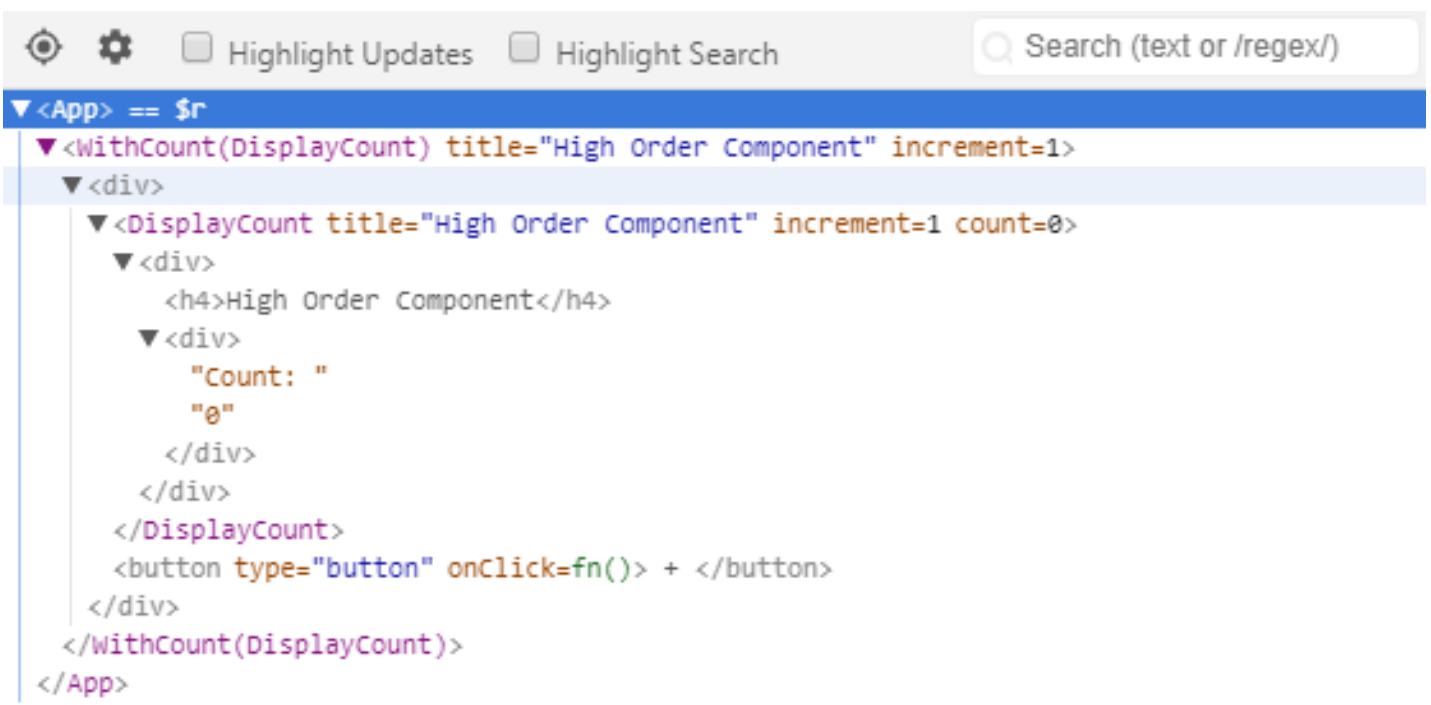
код компонента displayCount.tsx

код компонента withCount.tsx

Далее, опишем использование нашего компонента высшего порядка:

```
const Counter = withCount(DisplayCount);
/*
 * title - свойство напрямую передается компоненту DisplayCount
 * increment - свойство используется в компоненте высшего порядка
 */
const App = () => <Counter title="High Order Component" increment=
{1} /> ;
```

Итоговое дерево:



Свойства и состояние WithCount(DisplayCount):

```
Props
  increment: 1
  title: "High Order Component"
```

```
State
  count: 0
```

Свойства и состояние DisplayCount:

```
Props read-only
  count: 0
  increment: 1
  title: "High Order Component"
```

Здесь мы видим лишнее свойство `increment`, в случае необходимости от него можно избавиться, используя например метод omit в lodash.

3) **Ленивая загрузка компонентов:**
Для загрузки компонентов по требованию воспользуемся синтаксисом динамического импорта модулей.
В TypeScript этот синтаксис появился в версии 2.4.
Webpack, встречая динамический импорт, создает отдельный бандл для модулей, которые попадают под условие импорта.
Простейшее выражение для импорта:

```
import('module.ts').then((module) => {
    // элемент, который в модуле экспортируется по умолчанию, окажется в свойстве default
    const defaultExport = module.default;
    // остальные экспорты, например export function foo() {} - окажется в одноименном свойстве,
    // например module.foo
    const otherExport = module.OtherExport;
});
```

Далее мы напишем компонент, который принимает функцию,

возвращающую import, и выводит полученный компонент.
Создадим папку `lazy`, в ней компоненты `lazyComponent.tsx` и `lazyLoad.tsx`

LazyComponent — простой функциональный компонент, в реально приложении это может быть отдельная страница, или автономный виджет:

lazyComponent.tsx

LazyLoad — универсальный компонент для загрузки и вывода динамического компонента.
В случае необходимости прокидывать свойства в динамический компонент, LazyLoad можно переписать на компонент высшего порядка.

lazyLoad.tsx

Все-таки обновим настройки webpack, для возможности задавать бандлам имя:

webpack.config.js

И обновим настройки tsconfig.json — вручную укажем библиотеки, которые надо использовать TypeScript при компиляции. Нам нужна конкретно «es2015.promise», но для удобства добавим полный список по ES стандартам, и конечно DOM.

tsconfig.json

Использование компонента:

```
// webpackChunkName - имя для итогового бандла с динамическим модулем
// chunkFilename: '[name].bundle.js' создаст нам lazy-component.bundle.js
const load = () => import(/* webpackChunkName: 'lazy-component' */'./lazy/lazyComponent');
const App = ({title}: IAppProps) => <LazyLoad load={load} />;
```

4) **Render props**

Описание компонентов с render property в официальной документации React — по ссылке

Для удобства использования таких компонентов, обычно предоставляется несколько способов рендера.
Рассмотрим два основных: свойство **render** и свойство **children**.

Создадим папку `renderProps`, в ней компонент `displaySize.tsx` и компонент `windowQueries.tsx`

код компонента displaySize.tsx

код компонента windowQueries.tsx

Далее, опишем использование нашего компонента:

```
<WindowQueries>
    {({ width, height }) => <DisplaySize title="render children" width={width} height={height} />}
</WindowQueries>
```

```
<WindowQueries
    render={
        ({ width, height }) => <DisplaySize title="render property"
width={width} height={height} />
    }
/>
```

5) **Нюансы:**

Описание свойства children для доступа к дочерним элементам (не обязательно):

```
interface Props {
    children: React.ReactNode;
}
```

Описание свойства с JSX элементом, может использоваться для компонентов разметки:

```
interface Props {
    header: JSX.Element,
    body: JSX.Element
}

<Component
    header={<h1>Заголовок</h1>}
    body={<div>Содержимое</div>}
/>
```

Заключение

Мы создали окружение для разработки на React и TypeScript с минимально необходимыми настройками, и написали несколько простых компонентов.

TypeScript позволяет отказаться от использования PropTypes, и проверяет свойства компонентов во время разработки и компиляции (PropTypes же выдает ошибки только в запущенном приложении).

Такое преимущество строгой типизации, как автодополнение, распространяется и на JSX, а в файлах — декларациях библиотеки React вы можете быстро увидеть все возможные свойства для встроенных JSX элементов.

В сложных проектах использование TypeScript полностью оправдает себя — мы увидим это в таких моментах, как использование Redux (благодаря интерфейсам для вашего store), и работа с внешним API.

В статье №2 мы рассмотрим следующее:

1) Подключение Redux
2) Стандартные рецепты React, Redux и TypeScript
3) Работа с API
4) Production и development сборка проекта

В последующих статьях автор планирует описать: создание прогрессивного веб-приложения (PWA), серверный рендеринг, тестирование с Jest, и наконец оптимизацию приложения.

Автор просит прощения за не самое удачное оформление статьи, и повторно просит вносить свои предложения, по улучшению восприятия и читаемости этой статьи.

Благодарю за внимание!

Update 22.10.2017: Добавлен рецепт lazy load компонентов

Update 17.02.2018: Добавлен рецепт компонента с render property, обновлены зависимости (для устранения ошибок с типом ReactNode)

Похожие публикации

Архитектура модульных React + Redux приложений 2. Ядро

3

marshinov • 24 апреля 2017 в 10:04

Архитектура модульных React + Redux приложений

41

marshinov • 14 апреля 2017 в 03:27

Эволюция на React+Redux

28

Fen1kz • 7 апреля 2017 в 10:21

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации

2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ 40

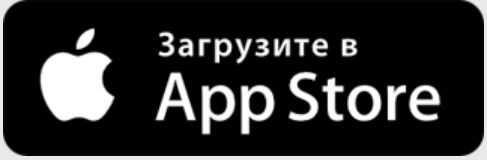
alizar • вчера в 15:14

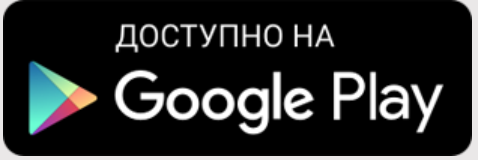
Стивен Хокинг и его работа — что дал ученый человечеству? 8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения 17

AlexeyNadezhin • вчера в 20:32





Полная версия