**ОГЛАВЛЕНИЕ**

|  |  |
| --- | --- |
| Глава 1. Обзор современных технологий создания клиентской части веб приложений | 1 |
| 1.1 Язык гипертекстовой разметки, таблицы стилей и препроцессоры | 2 |

**ВВЕДЕНИЕ**

bla

**ГЛАВА 1. ОБЗОР СОВРЕМЕННЫХ ТЕХНОЛОГИЙ РАЗРАБОТКИ КЛИЕНТСКОЙ ЧАСТИ ВЕБ ПРИЛОЖЕНИЙ**

* 1. **Язык гипертекстовой разметки*,* таблицы   
     стилей и препроцессоры**

При поверхностном рассмотрении любой веб страницы на просторах современного интернета, можно заключить вывод что любая клиентская часть приложений состоит из трёх основополагающих элементов:

1. *HTML-*документ;
2. Каскадные таблицы стилей;
3. Cкрипты динамического поведения.

По существу справедливым будет утверждение о том, что любое современное веб приложение состоит из этих трех элементов. Однако это не совсем так.

С постоянным увеличением количества ресурсов предоставляемых по средством сети интернет, неумолимо вставал вопрос о совершенствовании имеющихся механизмов и технологий. Однако прежде чем переходить к более современным средствам разработки следует разобраться что же из себя представляет основа любого приложения, а именно *html* (*Hyper Text Markup Language*) [1].

Итак, в том или ином виде основой любого веб приложения является *HTML*-документ. Именно на его основе браузер строит *DOM*-модель документа прежде чем отобразить результат на пользовательском мониторе. Однако данная технология является не столько языком, сколько стандартом, необходимым для стандартизации, разрабатываемых продуктов. Как и любой иной стандарт *HTML* пережил большое число изменений, связанных с появлением новых технологий, подходов и методик разработки. Так, если первоначально для создания документа для веб было достаточно одного языка гипертекстовой разметки, современное приложение уже строится на трех основополагающих элементах, взаимодействие которых и создает клиентское приложение в современном виде. Так, появление технологии каскадных таблиц стилей и постепенное расширение возможностей *CSS*, привело к устареванию большого числа первоначально используемых тегов таких как center, font и другие. На момент написания этой работы повсеместно используемым стандартом являлся *HTML5*.

Основным нововведением пятого стандарта семантические теги, теги управления аудио и видео содержимым, а также графический контейнер. Итак попробуем кратко разобраться что же привнесли данные элементы в процесс разработки.

Итак, благодаря большому числу семантических тегов, современное веб приложение можно разделить на логические единицы, которые явно указывают о своем назначении [2]. Такое нововведение в большой степени связанно развитием систем поиска и анализа данных. Так благодаря правильной комбинации и использованию семантических элементов можно оптимизировать работу поисковых систем, обернув значимую и второстепенную информацию в соответствующие теги. Так же правильное использование нововведений открывает большие возможности для устройств, предназначенных для людей с ограниченными возможностями. Так устройство авто чтения статьи может определить нужный текст и опустить рекламу.

Появление тегов управления медиа содержимым, вероятно связанно с расширением доли медиа контента. Появление данных механизмов сделало морально устаревшими более ранние технологии (Flesh), что уменьшило количество стороннего программного обеспечения.

Расширение возможностей графических библиотек (таких как *WebGl*), и их использования вероятно послужило необходимостью для введения специализированного тега.

Также на момент написания работы активно развивался шестой стандарт. Ключевой особенностью данного языка является введение пространств имен, а так же предполагает возможность создания одностраничных приложений без использования скриптов [3, 4].

Как было сказано выше многие теги *html* утратили свою необходимость в связи с развитием *CSS*. Данная технология представляет собою язык описания внешнего вида документа, написанного с помощью языка разметки [5]. Язык определяет внешний вид документа (шрифты, цвет текста и фона, отступы), а также позволяет создавать разметку [6, 7].

Уровень 1 (*CSS1*) [8]. Спецификация принята в 1997 году и определяет параметры шрифтов (размер, стиль, гарнитуру), цвета, атрибуты текста, выравнивания, свойства блоков (высота, внутренние и внешние отступы, рамки).

Уровень 2 (*CSS2*) [9]. Спецификация принята в 1998 году и расширила возможности первого уровня. Были добавлены блочная вёрстка, определение типов носителей, звуковые таблицы стилей, расширен механизм селекторов и прочее.

Уровень 2.1 (*CSS2.1*) [5]. Спецификация принята в 2011 году и в основном исправляла ошибки прошлой.

Уровень 3 (*CSS3*) [10]. Данная спецификация расширяет возможности предыдущих, а также добавляет возможности добавления анимации без использования *JavaScript*. Особенностью данной версии является модульная структура, то есть работа ведется по нескольким не связанным направлениям.

Так же активно ведется разработка нового четвертого уровня [11].

Однако использование *CSS* не позволяет использовать при написании таблиц стилей даже базовых возможностей программирования, таких как циклы и функции, механизмы наследования. Для увеличения уровня абстракции и расширения возможностей используются препроцессоры, такие как *SASS* [12] и *LESS* [13]. Главным преимуществом данных технологий является возможность повторного использования кода за счет модульной структуры и примесей, однако при этом перед использованием требуется компиляция кода.

Ещё один недостаток данного подхода заключается в сложной системе блочной-верстки. Из-за сложности системы для решения не тривиальных задач необходим более высокий уровень знаний, а также внимательность разработчика.

С целью совершенствования и упрощения блочной верстки предложены две технологии *CSS* *Flexbox* [14] и *Grid Layout* [15]. На момент написания данной работы данные технологии широко внедрялись, однако так как ограничения поддержки вынуждали разработчиков использовать две версии стилей.

Основным преимуществом использования технологии *Flexbox* является широкие возможности выравнивания элементов относительно осей координат. Можно легко выравнивать элементы по горизонтали и по вертикали, менять направление и порядок отображение элементов, растягивать блоки на всю высоту родителя или прибивать их к нижнему краю [14]. На Рисунке 1.1 показана ориентация элементов. На данный момент технология поддерживается всеми топовыми версиями браузеров.



Рисунок 1.1 – Ориентация элементов *flexbox*

Следует отметить что *JS-*реализация *flexbox* нашла свою нишу использования при разработке мобильных приложений. Данная технология широки используется в совокупности с *React Native*.

*Grid* представляет документ в виде сетки, благодаря чему можно легко манипулировать основными частями приложения. Основным преимуществом данной технологии является отсутствие зависимости отображения элементов вне зависимости от их порядка, что позволяет более эффективно разрабатывать интерфейсы для разных типов устройств. На данный момент ещё имеет недостаточную поддержку даже новыми браузерами.

В таблице 1.1 приведены данные по поддержке выше упомянутых технологий. В данной таблице приведены минимальные версии браузеров в которых реализована поддержка технологий [16, 17].

Таблица 1.1 – Поддержка *CSS* *Flexbox* и *Grid Layout*

|  |  |  |
| --- | --- | --- |
| Браузер | *CSS Flexbox* | *Grid Layout* |
| *Internet Explorer* | 11 (Не полностью) | 11 (Не полностью) |
| *Edge* | 16 | 16 |
| *Firefox* | 58 | 58 |
| *Chrome* | 49 | 63 |
| *Safari* | 11 | 11 |
| *iOS Safari* | 10.3 | 10.3 |
| *Opera Mini* | Все | – |
| *Chrome for Android* | 64 | 64 |
| *UC Browser for Android* | 11.8 | 11.8 |
| *Samsung Internet* | 4 | 6.2 |

**1.2**

Однако основой современного веб приложения является *JavaScript*. Данный язык программирования представляет собой мультипарадигменный язык программирования [16], направленный на создание клиентской части веб-приложения с использованием наиболее эффективных методик программирования нашего времени. В настоящее время *JS* позволяет использовать все преимущества объектно-ориентированного, императивного и функционального стилей программирования [16].

*JavaScript* предоставляет возможности в создании объектно-ориентированных клиентских приложений, однако в связи с реализацией других парадигм программирования, объектная модель имеет целый ряд особенностей по сравнению с другими языками программирования. Разнообразие предоставляемых средств предоставляемых средств делает язык гибким и позволяет осуществить быстрый переход с других языков программирования. Так же следует отметить что стандарт постоянно расширяют добавлением новых возможностей. Можно сказать, что данный язык постоянно совершенствуется и впитывает наиболее удачные идеи [17 – 19].

Первоначально синтаксис языка схож с синтаксисом С. Однако, с течением времени язык развивается и появляются определенные синтаксические особенности [16]: объекты, функции как объекты первого класса, автоматическое приведение типов, автоматическая сборка мусора, анонимные функции.

Структуру языка можно представить в виде трех частей [17, 20]:

1. Ядро (*ECMAScript*).
2. Объектная модель браузера (*Browser Object Model* или *BOM*).
3. Объектная модель документа (*Document Object Model* или *DOM*).

Ядро представляет собой основу для построения скриптового языка (*ECMAScript*). Описывает типы данных, инструкции, ключевые слова, операторы, объекты, регулярные выражения, не ограничивая возможности расширения функционала.

Объектная модель браузера ­представляет собой прослойку между ядром и объектной моделью документа [21]. При этом основная задача заключается в управлении окнами браузера и обеспечении их взаимодействия. Каждое из окон браузера представляется объектом *window*, центральным объектом *DOM* [22].

Помимо управления окнами *BOM* обычно обеспечивает следующие сущности [21]: управление фреймами, поддержка задержки в исполнении кода и зацикливания с задержкой, системные диалоги, управление адресом открытой страницы, управление информацией о браузере, управление информацией о параметрах монитора, ограниченное управление историей просмотра страниц, поддержка работы с HTTP cookie.

Объектная модель документа представляет собой интерфейс программирования для работы с XML и HTML документами [23]. В общем виде: генерация и добавление узлов, получение узлов, изменение узлов, изменение связей между узлами, удаление узлов.

В настоящее время *JavaScript* широко применяется при реализации:

* веб-приложений (клиентской части) – клиент-серверных программ, в котором клиентом является браузер, а сервером – веб-сервер, имеющих распределённую между сервером и клиентом логику [24];
* *AJAX* – популярном подходе к построению интерактивных пользовательских интерфейсов веб-приложений, заключающемся в «фоновом» асинхронном обмене данными браузера с веб-сервером [25, 26];
* *сomet* – широкое понятие, описывающее механизм работы веб-приложений, использующих постоянные *HTTP*-соединения, что позволяет веб-серверу отправлять данные браузеру без дополнительного запроса со стороны браузера [27];
* браузерные операционные системы – исходный код некоторых операционных систем более чем на половину состоит из *JS* [16];
* бурмарклеты – используется для создания небольших программ, размещаемых в закладки браузера [16];
* пользовательские скрипты в браузере – это программы, написанные на JavaScript, выполняемые в браузере пользователя при загрузке страницы и позволяющие автоматизировать часть работ, связанных с управлением пользовательским интерфейсом [16];
* серверные приложения – *JavaScript* может исполняться на серверах, использующих *Java* с 6 версии [28]. Наиболее популярные платформы: *Jaxer*, per*severe-framework*, *Helma*, *v8cgi*, *node.js* [29 – 31];
* виджеты – вспомогательная мини-программа для добавления дополнительных функций к пользовательскому интерфейсу;
* и др.

Однако не смотря на свою универсальность и динамику развития *JavaScript* имеет ряд недостатков. Во первых отсутствие строгой типизации приводит к неявным ошибкам из-за недочетов программиста. С целью исправления данной ситуации появился *TypeScript* [32, 33]. Данная технология вводит типизацию, позволяет определить ошибки на этапе компилирования (требует предварительной компиляции в *JS*), а также добавляет многие инструменты характерные для объектно-ориентированных технологий.

**ГЛАВА 2. КОМПОНЕНТНЫЙ ПОДХОД ПРОГРАММИРОВАНИЯ ВО FRONT-END**

**bla**

**ГЛАВА 3. РАЗРАБОТКА FRONT-END ПРИЛОЖЕНИЯ УПРАВЛЕНИЯ АВТОМАТИЗИРОВАННЫМ ТЕСТИРОВАНИЕМ ПРОЕКТОВ**

**bla**

**ЗАКЛЮЧЕНИЕ**

bla

**СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ**

*CSS* (*Cascading Style Sheets*)– Каскадные таблицы стилей.

*CSS Flexbox* (*Flexible Box Layout Module*) – модуль макета гибкого контейнера.

*DOM* (*Document Object Model*) – Документная объектная модель.

*HTML* (*Hyper Text Markup Language*) – Гипертекстовый язык разметки.

*JS* (*JavaScript*)

*LESS* (*Leaner Style Sheets*) – Линейные таблицы стилей.

*SASS* (*Syntactically Awesome Stylesheets*) – Превосходящие статические таблицы стилей.

**СПИСОК ТЕРМИНОВ**

*Back-end* – программно-аппаратной части сервиса.

*Cascading Style Sheets* (*CSS*) – формальный язык описания внешнего вида документа, написанного с использованием языка разметки.

*CSS Flexbox* – технология расположения блоков на странице, основанная на осях координат.

*DOM*-модель – объектная модель, используемая для XML/HTML-документов.

*Front-end* – клиентская сторона пользовательского интерфейса к программно-аппаратной части сервиса.

*CSS Grid layout* – технология двумерного макетирования в Веб, с возможностью размещения элементов в строках и столбцах.

*Hyper Text Markup Language* (*HTML*) – стандартизированный язык разметки документов во Всемирной паутине.

*Leaner Style Sheets* (*LESS*) – это динамический язык стилей, разработанный на основе технологии *SASS*.

*Syntactically Awesome Stylesheets* (*SASS*) – это метаязык на основе *CSS*, предназначенный для увеличения уровня абстракции *CSS* кода и упрощения файлов каскадных таблиц стилей.

*Web-based Graphics Library* (*WebGL*) – программная библиотека для языка программирования *JavaScript*, позволяющая создавать интерактивную *3D*-графику, функционирующую в широком спектре совместимых с ней веб-браузеров.

*React Native* – это *JS*-фреймворк для создания нативно отображаемых *iOS*- и *Android*-приложений.

**СПИСОК ИСПОЛЬЗОВАННОЙ ИСТОЧНИКОВ**

1. Википедия / HTML [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/HTML (дата обращения: 24.03.2018).
2. Goldstein, A. HTML5 & CSS3 For The Real World, 2nd Edition // A. Goldstein, L. Lazaris, E. Weyl – SitePoint, 2015. ­– 350 p.
3. ITVDN / Взгляд в HTML6 [Электронный ресурс]. – 2018. – Режим доступа: https://itvdn.com/ru/blog/article/looking-into-html6 (дата обращения: 24.03.2018).
4. Helix / HTML6 – когда необходима семантика [Электронный ресурс]. – 2017. – Режим доступа: http://helix.su/html6-kogda-neobhodima-semantika.html (дата обращения: 24.03.2018).
5. Википедия / CSS [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/CSS (дата обращения: 24.03.2018).
6. Сидерхолм, Д. CSS3 для веб-дизайнеров // Ден Сидерхолм – Манн, Иванов и Фербер, 2012. ­– 125 с.
7. Макфарланд, Д. Большая книга CSS3 // Д. Макфарланд – Питер, 2014. ­– 608 с.
8. W3C / Cascading Style Sheets, level 1 [Электронный ресурс]. – 2018. – Режим доступа: https://www.w3.org/TR/CSS1/ (дата обращения: 24.03.2018).
9. W3C / Cascading Style Sheets, level 2 CSS Specification [Электронный ресурс]. – 2017. – Режим доступа: https://www.w3.org/TR/2008/REC-CSS2-20080411/ (дата обращения: 24.03.2018).
10. W3C / Текущая работа CSS и как принять участие [Электронный ресурс]. – 2018. – Режим доступа: https://www.w3.org/Style/CSS/current-work (дата обращения: 24.03.2018).
11. WIRED / Discover What’s New in CSS 4 [Электронный ресурс]. – 2018. – Режим доступа: https://www.wired.com/2011/10/discover-whats-new-in-css-4/ (дата обращения: 24.03.2018).
12. SASS / Docs [Электронный ресурс]. – 2018. – Режим доступа: http://sass-lang.com/documentation/file.SASS\_REFERENCE.html (дата обращения: 24.03.2018).
13. LESS / Docs [Электронный ресурс]. – 2018. – Режим доступа: https://less-lang.info/ (дата обращения: 24.03.2018).
14. Про CSS / Flexbox [Электронный ресурс]. – 2018. – Режим доступа: http://css.yoksel.ru/flexbox/ (дата обращения: 24.03.2018).
15. CSS-TRICS / A Complete Guide to Grid [Электронный ресурс]. – 2018. – Режим доступа: https://css-tricks.com/snippets/css/complete-guide-grid/ (дата обращения: 24.03.2018).
16. Can I Use / Flexbox [Электронный ресурс]. – 2018. – Режим доступа: https://caniuse.com/#search=flexbox (дата обращения: 12.04.2018).
17. Can I Use / Grid Layout [Электронный ресурс]. – 2018. – Режим доступа: https://caniuse.com/#search=Grid%20layout (дата обращения: 12.04.2018).

**Приложение А. Исходный код некоторых компонент**

App.js:

import React, { Component } from 'react';

import { Router, Switch, Route } from 'react-router'

import createHistory from 'history/createBrowserHistory';

import Header from './components/Header';

import Footer from './components/Footer';

import HomePage from './pages/HomePage';

import AboutMePage from './pages/AboutMePage';

import ItemPage from './pages/ItemPage';

import ErrorPage from './pages/ErrorPage';

import LogoPage from './pages/LogoPage';

import TaskManagerPage from './pages/TaskManagerPage';

import \* as PropTypes from 'prop-types';

class App extends Component {

static PAGE\_LINKS = [

{ "link": "/task-manager", "title": "Task manager" },

{ "link": "/about", "title": "About me" }

];

getChildContext() {

return {

pageLinks: App.PAGE\_LINKS

}

}

render() {

return (

<div className="app">

<Header text="Front-end EPAM training" />

<Router history={ createHistory() }>

<Switch>

<Route exact path="/" component={ HomePage } />

<Route path="/about" component={ AboutMePage } />

<Route path="/task-manager" component={ TaskManagerPage } />

<Route path="/item/:name/:environment" component={ ItemPage } />

<Route path="/logo.txt" component={ LogoPage }/>

<Route path="/\*" component={ ErrorPage } />

</Switch>

</Router>

<Footer date="2017" title="Kuzmiankou Anatoli" />

</div>

);

}

static childContextTypes = {

pageLinks: PropTypes.arrayOf(PropTypes.shape({

link: PropTypes.string,

title: PropTypes.string

}))

}

}

export default App;

TaskManagerPage.js:

import React, { Component } from 'react';

import AddComponentWindow from '../components/AddComponentWindow';

import TableBox from '../components/TableBox';

import TaskComponentManager from '../components/TaskComponentManager';

import Counter from '../components/Counter';

import Main from '../components/Main';

import \* as PropTypes from 'prop-types';

import { connect } from 'react-redux'

import { loadTasks, loadTasksSuccess } from '../redux/action/TaskAction';

class TaskManagerPage extends Component {

componentDidMount() {

this.props.loadingData();

}

createTaskManager(data, number) {

const { name, environments } = data;

return (

<TaskComponentManager taskName={ name } key={ name } tasks={ environments } />

);

}

static ENVIRONMENTS = [

{ name: "int", title: "INT" },

{ name: "qa", title: "QA" },

{ name: "staging", title: "Staging" },

{ name: "production", title: "Production" }

];

getChildContext() {

return {

environments: TaskManagerPage.ENVIRONMENTS

};

}

render() {

const { loading, tasks } = this.props;

return (

<Main name="Task manager" loading={ loading }>

<Counter title="Shared Services / Component" name="components-count" count={ tasks.length } />

<AddComponentWindow />

<TableBox>

{ tasks.map( (item, number) => this.createTaskManager(item, number) ) }

</TableBox>

</Main>

);

}

static childContextTypes = {

environments: PropTypes.arrayOf(PropTypes.shape({

name: PropTypes.string,

title: PropTypes.string

}))

}

}

const mapStateToProps = state => {

const { TaskReducer } = state;

return { ...TaskReducer };

}

const mapDispatchToProps = dispatch => ({

loadingData: () => {

dispatch(loadTasks());

fetch('http://localhost:9999/data/projects')

.then(res => res.json())

.then(json => dispatch(loadTasksSuccess(json)));

}

});

export default connect(mapStateToProps, mapDispatchToProps)(TaskManagerPage);

Task.js:

import \* as BodyFactory from "./lib/TaskBodyFactory";

import \* as StatusFactory from "./lib/TaskStatusFactory";

import TaskStatus from "./lib/TaskStatus";

import \* as PropTypes from 'prop-types';

import { connect } from 'react-redux';

import { refreshTask, refreshTaskSuccess } from '../../redux/action/TaskAction';

import autoBind from 'react-autobind';

class Task extends Component {

constructor(props) {

super(props);

autoBind(this);

}

refreshAction() {

const { name, env } = this.props;

this.props.refreshAction({ name, env });

}

shouldComponentUpdate(nextProps, nextState) {

for(let index in this.props) {

if(this.props[index] !== nextProps[index]) {

return true;

}

}

return false;

}

render() {

const { env, link, data } = this.props;

const { status, version } = this.props.data;

return(

<td className={ `task task-${ env } task-${ TaskStatus[status].toLowerCase() }` }>

<div className="header">

<div className="version">{ version }</div>

{ StatusFactory.createStatus(status) }

</div>

{ BodyFactory.createBodyElem(data) }

<div className="control-panel">

<a href={ link } className="link" ><span className="icon-link" /></a>

<button className="refresh icon-refresh" onClick={ this.refreshAction } />

</div>

</td>

);

}

static defaultProps = {

name: "PROJECT",

env: "int",

data: {

version: "v-.-.-",

status: TaskStatus.MESSING

}

}

static propsTypes = {

name: PropTypes.string,

env: PropTypes.string,

data: PropTypes.shape({

version: PropTypes.string,

status: PropTypes.number,

timestamps: PropTypes.string,

testResult: PropTypes.shape({

total: PropTypes.number,

failed: PropTypes.number,

passed: PropTypes.number,

skipped: PropTypes.number

}),

logo: PropTypes.string

}),

link: PropTypes.string

};

}

const mapDispatchToProps = dispatch => ({

refreshAction: params => {

dispatch(refreshTask());

fetch('http://localhost:9999/data/refresh', {

method: 'post',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json'

},

body: JSON.stringify(params)

}).then(res => res.json())

.then(data => dispatch(refreshTaskSuccess(params, data)));

}

});

export default connect(null, mapDispatchToProps)(Task);

TaskBodyFactory.js:

import React from 'react';

import Status from './TaskStatus';

import moment from 'moment';

const viewQueue = status => (

<div className="inqueue" key="inqueue">In queue</div>

)

const viewTests = tests => (

<div key="tests">

{ `${ tests.failed } failed, ${ tests.passed } passed, ${ tests.skipped } skipped, ${ tests.total } total` }

</div>

);

const viewLogo = link => (

<div key="link">See <a href={ `/${ link }` } target="\_blank">{ link }</a></div>

);

const viewTime = time => (

<div key="date">{ moment(time).fromNow() }</div>

);

export const createBodyElem = data => (

<div className="body">

{ data.status === Status.QUEUE && viewQueue(data.status) }

{ data.testResult && viewTests(data.testResult) }

{ data.logo && viewLogo(data.logo) }

{ data.timestamps && viewTime(data.timestamps) }

</div>

);

TaskStatus.js:

const TaskStatus = {

QUEUE: 'QUEUE',

MISSING: 'MISSING',

RUNNING: 'RUNNING',

FAILED: 'FAILED',

SUCCESS: 'SUCCESS',

W\_O\_FAILED: 'W\_O\_FAILED'

}

export default TaskStatus;

TaskStatusFactory.js:

import React from 'react';

import Status from './TaskStatus';

const titles = {

[Status.QUEUE]: "IN QUEUE",

[Status.MISSING]: "MISSING AURA.JSON",

[Status.RUNNING]: "RUNNING",

[Status.FAILED]: "FAILED",

[Status.W\_O\_FAILED]: "FAILED"

}

const icons = {

[Status.SUCCESS]: "icon-ok",

[Status.W\_O\_FAILED]: "icon-warning"

}

export const createStatus = status => (

<div className={ `status ${ icons[status] }` }>{ titles[status] }</div>

);

AddComponentWindow.js:

import React, {Component} from 'react';

import TextForm from './TextForm';

import ModalWindow from './ModalWindow';

import \* as PropTypes from 'prop-types';

import { addProjectTasks, addProjectTasksSuccess } from '../redux/action/TaskAction';

import { connect } from 'react-redux';

import autoBind from 'react-autobind';

class AddComponentWindow extends Component {

constructor(props) {

super(props);

autoBind(this);

}

getChildContext() {

const { environments } = this.context;

const fields = [{ name: "name", labelValue: "Item name" }];

environments && environments.forEach( item =>

fields.push({

name: item.name,

labelValue: `${ item.title } envirment url`

})

);

return { fields };

}

formAction(data) {

this.hideWindow();

this.props.addComponent(data);

this.form.resetAction();

}

hideWindow() {

this.modalWindow.hide();

}

showWindow() {

this.modalWindow.show();

}

render() {

return (

<figure className="add-component" name="add-component">

<button name="add-component" onClick={ this.showWindow }

className="btn-add-component">Add Component</button>

<ModalWindow ref={ window => this.modalWindow = window } title="Add component item">

<TextForm ref={ form => this.form = form } submitName="Add" submitAction={ this.formAction }/>

</ModalWindow>

</figure>

);

}

static contextTypes = {

environments: PropTypes.arrayOf(PropTypes.shape({

name: PropTypes.string,

title: PropTypes.string

})),

};

static childContextTypes = {

fields: PropTypes.arrayOf(PropTypes.shape({

name: PropTypes.string,

labelValue: PropTypes.string

}))

};

}

const mapDispatchToProps = dispatch => ({

addComponent: params => {

dispatch(addProjectTasks(params.name));

fetch('http://localhost:9999/data/add-project', {

method: 'post',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json'

},

body: JSON.stringify(params)

}).then(res => res.json())

.then(json => dispatch(addProjectTasksSuccess(json)));

}

});

export default connect(null, mapDispatchToProps)(AddComponentWindow);

ModalWindow.js:

import React, {Component} from "react";

import \* as PropTypes from 'prop-types';

import autoBind from 'react-autobind';

class ModalWindow extends Component {

constructor(props) {

super(props);

autoBind(this);

}

componentDidMount() {

if(this.props.startVisible) {

this.show();

}

else {

this.hide();

}

}

hide() {

this.window.style.display = "none";

}

show() {

this.window.style.display = "block";

}

render() {

const { id, name, title, children } = this.props;

return (

<figure

id={ id }

name={ name }

className={ "modal modal-" + name }

ref={ window => this.window = window }

>

<div className="modal-content">

<h2>{ title }</h2>

<button className="btn-close" onClick={ this.hide }>x</button>

{ children }

</div>

</figure>

);

}

static defaultProps = {

name: "window",

title: "Modal window",

startVisible: false

};

static propTypes = {

id: PropTypes.string,

name: PropTypes.string,

title: PropTypes.string,

startVisible: PropTypes.bool

};

}

export default ModalWindow;

TaskComponentManager.js:

import React, {Component} from 'react';

import Task from './task/Task';

import \* as PropTypes from 'prop-types';

import { connect } from 'react-redux';

import { removeProjectTasks, removeProjectTasksSuccess } from '../redux/action/TaskAction';

import autoBind from 'react-autobind';

class TaskComponentManager extends Component {

constructor(props) {

super(props);

autoBind(this);

}

createTask(key, task) {

const name = this.props.taskName;

return (

<Task

name={ name }

key={ key }

env={ key }

data={ task }

link={ `/item/${ name }/${ key }` }

/>

)

}

createEmptyTask(key) {

return <td key={ key }><div className="task-empty" /></td>

}

createTasks(tasks) {

return this.context.environments.map(item => {

const name = item.name;

const task = tasks[name];

return task ? this.createTask(name, task) : this.createEmptyTask(name);

});

}

removeComponent() {

this.props.removeAction({ name: this.props.taskName });

}

render() {

const { name, taskName, tasks } = this.props;

return (

<tbody className={ `task-component-manager manager-${ name }` }>

<tr>

<td className="task-manager">

<h2 className="title">{ taskName }</h2>

<ul className="settings icon-setting">

<li onClick={ this.removeComponent }>Remove</li>

</ul>

</td>

{ this.createTasks(tasks) }

</tr>

</tbody>

);

}

static defaultProps = {

name: "task-manager"

};

static contextTypes = {

environments: PropTypes.arrayOf(PropTypes.shape({

name: PropTypes.string,

title: PropTypes.string

})),

};

static propTypes = {

name: PropTypes.string,

taskName: PropTypes.string,

tasks: PropTypes.objectOf(PropTypes.shape({

version: PropTypes.string,

status: PropTypes.string,

testResult: PropTypes.shape({

total: PropTypes.number,

failed: PropTypes.number,

passed: PropTypes.number,

skipped: PropTypes.number

}),

timestatmps: PropTypes.string,

logo: PropTypes.string

}))

};

}

const mapDispatchToProps = dispatch => ({

removeAction: params => {

dispatch(removeProjectTasks());

fetch('http://localhost:9999/data/remove-tasks', {

method: 'post',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json'

},

body: JSON.stringify(params)

}).then(res => res.json())

.then(data => dispatch(removeProjectTasksSuccess(params.name, data.result)));

}

});

export default connect(null, mapDispatchToProps)(TaskComponentManager);

NavBar.js:

import React from 'react';

import \* as PropTypes from 'prop-types';

const NavBar = (props, context) => {

const { name } = context

return (

<nav className="nav-bar">

<ul className="nav-bar-menu">

<li className="horisontal"><a href="/">Home</a></li>

{

(name)

? <li className="horisontal">{ name }</li>

: undefined

}

</ul>

</nav>

);

}

NavBar.contextTypes = {

name: PropTypes.string

};

export default NavBar;

TableBox.js:  
import React, {Component} from 'react';

import \* as PropTypes from 'prop-types';

class TableBox extends Component {

createTH(name) {

return (

<th className="column-tittle" key={ name }>{ name }</th>

);

}

render() {

const { environments } = this.context;

return (

<table className="table-box">

<thead>

<tr>

{ this.createTH("Name") }

{ environments && environments.map( item => this.createTH(item.title) ) }

</tr>

</thead>

{ this.props.children }

</table>

)

}

static contextTypes = {

environments: PropTypes.arrayOf(PropTypes.shape({

title: PropTypes.string

})),

};

}

export default TableBox;

Main.js:

import React, { Component } from 'react';

import \* as PropTypes from 'prop-types';

import NavBar from './NavBar';

import Loading from './Loading';

class Main extends Component {

getChildContext() {

const { name } = this.props;

return {

name

}

}

render() {

const { loading, children } = this.props;

return (

<main>

<NavBar />

{

(loading)

? <Loading />

: <div className="main-content">{ children }</div>

}

</main>

);

}

static childContextTypes = {

name: PropTypes.string,

loading: PropTypes.bool

};

}

export default Main;