Технологии разработки мобильных приложений. Практикум

Составитель: Салех Х.М.

1. Javascript, Webpack, JQuery

Введение

JavaScript — прототипно-ориентированный сценарный язык программирования. Является реализацией языка ECMAScript. JavaScript обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.Основные архитектурные черты: динамическая типизация, слабая типизация, автоматическое управление памятью, прототипное программирование, функции как объекты первого класса.

Webpack можно охарактеризовать, как “сборщик модулей”. Он берет JavaScript модули с необходимыми зависимостями, и затем соединяет их вместе как можно более эффективным способом, на выходе создавая единый JS-файл. На первый взгляд – ничего особого, не так ли? Например, такие инструменты, как RequreJS позволяют делать подобные вещи вот уже много лет.

Весь трюк заключается вот в чем. Webpack не ограничен в использовании только JavaScript модулей. Применяя специальные Загрузчики, Webpack понимает, что JavaScript модулям могут потребоваться для их работы, например, CSS файлы, а им, в свою очередь, изображения. При этом, результат работы Webpack будет содержать только те ресурсы, которые действительно нужны для работы приложения.

jQuery — библиотека JavaScript, фокусирующаяся на взаимодействии JavaScript и HTML. Библиотека jQuery помогает легко получать доступ к любому элементу DOM, обращаться к атрибутам и содержимому элементов DOM, манипулировать ими. Также библиотека jQuery предоставляет удобный API для работы с AJAX.

Настройка сборки проекта

1. Установить node

2. Создать папку с проектом и выполнить в ней npm init

3. Добавить скрипт в секцию scripts в package.json:

```
"start": "node ./node_modules/webpack-dev-server/bin/webpack-dev-server.js"
```

4. Устанавливаем нужные зависимости:

```
npm install --save webpack webpack-dev-server jquery style-loader css-loader
```

5. Создать файл index.js со следующим содержимым:

```
console.log('Hello World!')
```

6. Создать пустой файл app.css и файл index.html со следующим содержимым:

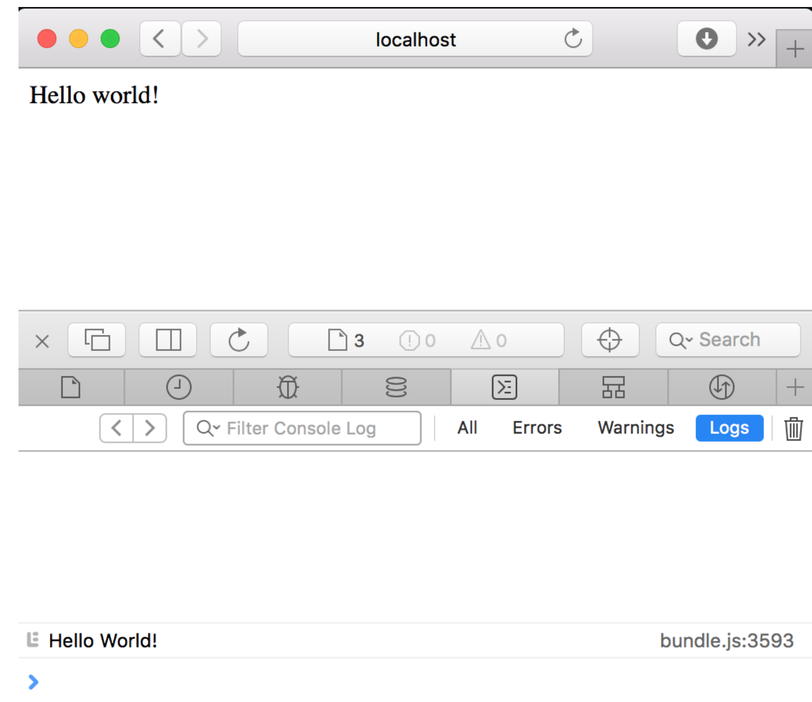
```
<html>
  <body>
    Hello world!
    <script src="public/bundle.js"></script>
  </body>
</html>
```

6. Создать файл webpack.config.js со следующим содержимым:

```
const webpack = require('webpack');
module.exports = {
  entry: [
    './index.js'
  ],
  output: {
    path: __dirname + '/public',
    publicPath: '/public/',
    filename: 'bundle.js'
```

```
    },
    module: {
      loaders: [
        { test: /\.css$/, loaders: ['style-loader', 'css-
loader'] }
      ]
    }
  };
```

7. Запускаем npm start и проверяем http://localhost:8080:



Создания приложения для управления списком задач

Поменять index.html:

```
<html>
 <body>
  <h1>Todo List</h1>

  <ul id='todos'></ul>

  <input id='title' />
  <button id='add-todo'>Add</button>
```

```
    <script src="public/bundle.js"></script>
  </body>
</html>
```

Поменять index.js:

```
const $ = require('jquery');
require('./app.css');
$(document).ready(function() {
  $('#add-todo').on('click', function() {
    var title = $('#title').val();
    $('#todos').append('<li>' + title + '</li>');
  });
});
```

В результате получаем приложение, способное добавлять задания в список.

Общие задания:

1. Подключить ES7 с помощью babel
2. Подключить загрузчик для LESS/SASS
3. Очищать поле ввода после добавления задания
4. Выдавать предупреждение при попытке добавить задание с пустым названием
5. Добавлять тень при событии hover на элементе списка

Варианты индивидуальных заданий:

1. Удаление задания при нажатии на кнопку рядом с элементом списка, после удаления элемент исчезает, последнее задание удалять нельзя
2. Удаление задания при нажатии на кнопку рядом с элементом списка, после удаления элемент зачеркивается и перемещается вниз списка, а кнопка исчезает

3. Перемещение задания вверх либо вниз при нажатии на кнопки рядом с элементом списка, после перемещения – добавить тень на 3 секунды, невозможные перемещения - игнорировать
4. Перемещение задания вверх либо вниз при нажатии на кнопки рядом с элементом списка, у первого и последнего элемента соответствующие кнопки должны исчезать
5. Вывод времени добавления задания момента серыми цветом, кнопка «выполнить» делает текст зеленым и выводит затраченное на выполнение задание время в секундах
6. Вывод времени добавления задания момента серыми цветом; задания, не выполненные за 10 секунд должны подсвечиваться красным
7. Пометка задания приоритетным (цвет текста - красный) при нажатии на кнопку рядом с элементом списка, повторное нажатие убирает приоритет
8. Пометка задания приоритетным (цвет текста - красный) и перенос его наверх списка при нажатии на кнопку рядом с элементом списка

2. VueJS

Введение

Vue — это прогрессивный фреймворк для создания пользовательских интерфейсов. В отличие от фреймворков-монолитов, Vue создан пригодным для постепенного внедрения. Его ядро в первую очередь решает задачи уровня представления (view), что упрощает интеграцию с другими библиотеками и существующими проектами. С другой стороны, Vue полностью подходит и для создания сложных одностраничных приложений (SPA, Single-Page Applications), если использовать его совместно с современными инструментами и дополнительными библиотеками.

Настройка сборки проекта

1. Скопировать папку проекта из первой лабораторной работы и переименовать ее

2. Устанавливаем Vue (актуальная версия на момент написания данной работы – 2.1.10):

```
npm install --save vue
```

Написание первого приложения на Vue

В ядре Vue.js находится система, которая позволяет декларативно отображать данные в DOM, используя простые шаблоны.

Поменять index.html:

```
<html>
  <body>
    <div id='app'>
      {{message}}
    </div>

    <script src="public/bundle.js"></script>
  </body>
</html>
```

Поменять index.js:

```
var Vue = require('vue/dist/vue.js');

var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
});
```

В результате на странице будет выводиться текст Hello Vue! Каждый vm — это корневой экземпляр Vue, создаваемый функцией-конструктором, в нем есть несколько полезных секций:

- el – селектор корневого элемента
- data – описание данных
- methods – описание методов

Правила отрисовки html задаются с помощью интерполяции (фигурные скобки) и директив, таких как v-for, v-if и других. Обработка событий производится с помощью директивы v-on. Больше сведений можно найти в официальной документации на https://vuejs.org либо https://ru.vuejs.org

Создания приложения для управления списком задач

Поменять index.html:

```
<html>
  <body>
    <div id='app'>
      <h1>Todo List</h1>

      <ul>
        <li v-for='todo in todos'>
          {{todo.text}}
        </li>
      </ul>

      <input v-model='newTodo' />
      <button v-on:click='addTodo'>Add</button>
    </div>

    <script src="public/bundle.js"></script>
```

```
  </body>
</html>
```

Поменять index.js:

```
var Vue = require('vue/dist/vue.js');

var app = new Vue({
  el: '#app',
  data: {
    newTodo: null,
    todos: []
  },
  methods: {
    addTodo: function() {
      this.todos.push({
        text: this.newTodo
      });
    }
  }
});
```

В результате получаем приложение, способное добавлять задания в список.

Использование компонентов

Другой важной концепцией Vue являются компоненты. Эта абстракция позволяет собирать большие приложения из меньших кусочков. Компоненты представляют собой пригодные к повторному использованию объекты. Во Vue, компонент — это, по сути, экземпляр Vue с предустановленными опциями. В крупных приложениях разделение на компоненты становится обязательным условием для сохранения управляемости процесса разработки.

Передача свойств в компоненты производится с помощью директивы v-bind.

Поменять index.html:

```
<html>
  <body>
    <div id='app'>
      <h1>Todo List</h1>

      <ul>
        <todo-item v-for='todo in todos' v-
bind:todo='todo' />
      </ul>

      <todo-input v-on:click='addTodo' />
    </div>

    <script src="public/bundle.js"></script>
  </body>
</html>
```

Поменять index.js:

```
var Vue = require('vue/dist/vue.js');

Vue.component('todo-item', {
  props: ['todo'],
  template: `
    <li>{{todo.text}}</li>
  `
});
```

```
Vue.component('todo-input', {
  data: function() {
    return { newTodo: null };
  },
  methods: {
    onClick: function() {
      this.$emit('click', this.newTodo);
    }
  },
  template: `
    <div>
      <input v-model='newTodo' />
      <button v-on:click='onClick'>Add</button>
    </div>
  `
});

var app = new Vue({
  el: '#app',
  data: {
    todos: []
  },
  methods: {
    addTodo: function(text) {
      this.todos.push({
        text: text
      });
    }
  }
});
```

Общие задания:

1. JQuery не используется
2. Очищать поле ввода после добавления задания
3. Выдавать предупреждение при попытке добавить задание с пустым названием
4. Выдавать предупреждение при попытке добавить задание с текстом, совпадающим с уже существующим
5. Добавить компонент, выводящий количество заданий в списке
6. Не допускать добавление более десяти заданий, при достижении максимума – поле ввода и кнопка должны становиться неактивными

Варианты индивидуальных заданий:

1. Удаление задания при нажатии на кнопку рядом с элементом списка, после удаления элемент исчезает, последнее задание удалять нельзя
2. Удаление задания при нажатии на кнопку рядом с элементом списка, после удаления элемент зачеркивается и перемещается вниз списка, а кнопка исчезает
3. Перемещение задания вверх либо вниз при нажатии на кнопки рядом с элементом списка, после перемещения – добавить тень на 3 секунды, невозможные перемещения - игнорировать
4. Перемещение задания вверх либо вниз при нажатии на кнопки рядом с элементом списка, у первого и последнего элемента соответствующие кнопки должны исчезать
5. Вывод времени добавления задания момента серыми цветом, кнопка «выполнить» делает текст зеленым и выводит затраченное на выполнение задание время в секундах
6. Вывод времени добавления задания момента серыми цветом; задания, не выполненные за 10 секунд должны подсвечиваться красным

7. Пометка задания приоритетным (цвет текста - красный) при нажатии на кнопку рядом с элементом списка, повторное нажатие убирает приоритет
8. Пометка задания приоритетным (цвет текста - красный) и перенос его наверх списка при нажатии на кнопку рядом с элементом списка

3. React

Настройка сборки проекта

1. Скопировать папку проекта из первой лабораторной работы и переименовать ее

2. Устанавливаем React:

```
    npm install --save react react-dom babel-preset-react
babel-preset-stage-1
```

3. Создаем в корневом каталоге файл `.babelrc` со следующим содержимым:

```
{
  "presets": ["react", "stage-1"]
}
```

Написание первого приложения

Поменять index.html:

```
<html>
  <body>
    <div id='app' />

    <script src="public/bundle.js"></script>
  </body>
</html>
```

Поменять index.js:

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

ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('app')
);
```

В результате на странице будет выводиться текст Hello, world!

Создание первого компонента

Создать в корневом каталоге папку components и в ней создать файл Message.js со следующим содержимым:

```
import React from 'react';

const Message = ({text}) => <h1>{text}</h1>;

export default Message;
```

Поменять index.js:

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

import Message from './components/Message';

ReactDOM.render(
  <Message text='Hello, world!' />,
  document.getElementById('app')
);
```

В результате на странице будет выводиться текст Hello, world!

Создания приложения для управления списком задач

Поменять index.js:

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

```
import App from './containers/App';

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

Создать папку containers и затем создать в ней файл App.js со следующим содержимым:

```
import React, {Component} from 'react';
import TodoItem from '../components/TodoItem';
import TodoInput from '../components/TodoInput';

class App extends Component {
  state = { todos: [] };

  addTodo = text => {
    let todos = this.state.todos.concat({text});
    this.setState({todos});
  }

  render() {
    return (
      <div>
        <h1>Todo List</h1>

        {
          this.state.todos.map(todo =>
            <ul key={todo.text}>
              <TodoItem text={todo.text} />
            </ul>
```

```
            )
          }

          <TodoInput onSubmit={this.addTodo} />
        </div>
      );
    }
  }

export default App;
```

Создать файл components/TodoItem.js со следующим содержимым:

```
import React from 'react';

const TodoItem = ({text}) => <li>{text}</li>;

export default TodoItem;
```

Создать файл components/TodoInput.js со следующим содержимым:

```
import React, {Component} from 'react';

class TodoInput extends Component {
  state = { newTodo: '' };

  changeText = evt => this.setState({ newTodo:
evt.target.value });

  handleAdd = () =>
this.props.onSubmit(this.state.newTodo);

  render() {
    return (
```

```
        <div>
          <input onChange={this.changeText}/>
          <button onClick={this.handleAdd}>Add</button>
        </div>
      );
    }
  }

  export default TodoInput;
```

В результате получаем приложение, способное добавлять задания в список.

Общие задания:

1. JQuery не используется
2. Очищать поле ввода после добавления задания
3. Выдавать предупреждение при попытке добавить задание с пустым названием
4. Выдавать предупреждение при попытке добавить задание с текстом, совпадающим с уже существующим
5. Добавить компонент, выводящий количество заданий в списке
6. Не допускать добавление более десяти заданий, при достижении максимума – поле ввода и кнопка должны становиться неактивными

Варианты индивидуальных заданий:

1. Удаление задания при нажатии на кнопку рядом с элементом списка, после удаления элемент исчезает, последнее задание удалять нельзя

2. Удаление задания при нажатии на кнопку рядом с элементом списка, после удаления элемент зачеркивается и перемещается вниз списка, а кнопка исчезает

3. Перемещение задания вверх либо вниз при нажатии на кнопки рядом с элементом списка, после перемещения – добавить тень на 3 секунды, невозможные перемещения - игнорировать

4. Перемещение задания вверх либо вниз при нажатии на кнопки рядом с элементом списка, у первого и последнего элемента соответствующие кнопки должны исчезать

5. Вывод времени добавления задания момента серыми цветом, кнопка «выполнить» делает текст зеленым и выводит затраченное на выполнение задание время в секундах

6. Вывод времени добавления задания момента серыми цветом; задания, не выполненные за 10 секунд должны подсвечиваться красным

7. Пометка задания приоритетным (цвет текста - красный) при нажатии на кнопку рядом с элементом списка, повторное нажатие убирает приоритет

8. Пометка задания приоритетным (цвет текста - красный) и перенос его наверх списка при нажатии на кнопку рядом с элементом списка

4. Redux

Введение

Redux позиционирует себя как предсказуемый контейнер состояния (state) для JavaScript приложений. Редакс вдохновлен Flux и Elm. Redux предлагает думать о приложении, как о начальном состоянии модифицируемом последовательностью действий (actions), что я считаю действительно хорошим подходом для сложных веб-приложений, открывающим много возможностей.

Настройка сборки проекта

1. Скопировать папку проекта из третьей лабораторной работы и переименовать ее

2. Устанавливаем React:

```
npm install --save redux react-redux
```

Перевод приложения на работу с redux-хранилищем

Изменяем index.js:

```
import React from 'react';
import ReactDOM from 'react-dom';
import reducers from './reducers';

import {Provider} from 'react-redux';
import { createStore, combineReducers } from 'redux';

import App from './containers/App';

const combinedReducers = combineReducers(reducers);
const store = createStore(combinedReducers);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('app')
);
```

Изменяем файл containers/App.js для работы с хранилищем:

```
import React, {Component} from 'react';
import { connect } from 'react-redux';
import { bindActionCreators } from 'redux';

import TodoItem from '../components/TodoItem';
```

```
import TodoInput from '../components/TodoInput';

import addTodo from '../actions/add_todo';

class App extends Component {
  render() {
    return (
      <div>
        <h1>Todo List</h1>

        {
          this.props.todos.map(todo =>
            <ul key={todo.text}>
              <TodoItem text={todo.text} />
            </ul>
          )
        }

        <TodoInput onSubmit={this.props.addTodo} />
      </div>
    );
  }
}

const mapStateToProps = state => ({
  todos: state.todos
});

const mapDispatchToProps = dispatch => bindActionCreators(
  {addTodo},
```

```
    dispatch
  );

  export default connect(mapStateToProps,
mapDispatchToProps)(App);
```

Создаем папку actions и помещаем туда файл add_todo.js со следующим содержимым:

```
  export default text => ({
    type: 'ADD_TODO',
    text
  });
```

Создаем папку reducers и помещаем туда файл index.js со следующим содержимым:

```
  const initialState = [];

  const todosReducer =  (state = initialState, action) => {
    switch (action.type) {
    case 'ADD_TODO':
      return state.concat({text: action.text});
    default:
      return state;
    }
  };
  export default {todos: todosReducer};
```

Асинхронные действия и redux-thunk

Устанавливаем зависимость:

```
npm install --save redux-thunk
```

Изменяем файл index.js:

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

import {Provider} from 'react-redux';
import thunk from 'redux-thunk';
import { createStore, combineReducers, applyMiddleware }
from 'redux';

import App from './containers/App';
import reducers from './reducers';

const combinedReducers = combineReducers(reducers);
const store = createStore(
  combinedReducers,
  applyMiddleware(thunk)
);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('app')
);
```

Изменяем файл actions/add_todo.js, теперь мы можем вызывать функцию dispatch несколько раз и создавать несколько действий:

```
export default text => dispatch => {
```

```
    dispatch({
      type: 'ADD_TODO',
      text
    });
  };
```

Требования:

1. Вся работа с данными – через redux
2. Допускается создание вложенных редьюсеров

Реализовать следующие действия:

1. Очистка списка по нажатию кнопки
2. Добавить текстовое поле для поиска по подстроке

Варианты индивидуальных заданий:

1. Удаление задания при нажатии на кнопку рядом с элементом списка, после удаления элемент исчезает, последнее задание удалять нельзя
2. Удаление задания при нажатии на кнопку рядом с элементом списка, после удаления элемент зачеркивается и перемещается вниз списка, а кнопка исчезает
3. Перемещение задания вверх либо вниз при нажатии на кнопки рядом с элементом списка, после перемещения – добавить тень на 3 секунды, невозможные перемещения - игнорировать
4. Перемещение задания вверх либо вниз при нажатии на кнопки рядом с элементом списка, у первого и последнего элемента соответствующие кнопки должны исчезать
5. Вывод времени добавления задания момента серыми цветом, кнопка «выполнить» делает текст зеленым и выводит затраченное на выполнение задание время в секундах
6. Вывод времени добавления задания момента серыми цветом; задания, не выполненные за 10 секунд должны подсвечиваться красным

7. Пометка задания приоритетным (цвет текста - красный) при нажатии на кнопку рядом с элементом списка, повторное нажатие убирает приоритет

8. Пометка задания приоритетным (цвет текста - красный) и перенос его наверх списка при нажатии на кнопку рядом с элементом списка

5. Работа с API

Создание API

1. Создать папку с проектом сервера и выполнить в ней npm init
2. Добавить скрипт в секцию scripts в package.json:

```
"start": "node ./index.js"
```

3. Устанавливаем нужные зависимости:

```
npm install --save express body-parser
```

4. Создать файл index.js со следующим содержимым:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.urlencoded({ extended: false }))

app.get('/', (req, res) => {
  res.send(`You requested list of todos`);
})

app.post('/', (req, res) => {
  const body = req.body.Body;
  res.set('Content-Type', 'text/plain');
  res.send(`You sent: ${body} to Express`);
})
```

```
app.listen(3000, err => {
  if (err) {
    throw err;
  }

  console.log('Server started on port 3000');
})
```

После запуска при посещении localhost:3000 должно отображаться сообщение.

Работа с API из клиентского приложения

Для выполнения запросов предлагается использовать библиотеку axios. К примеру, action для загрузки списка задач может выглядеть следующим образом:

```
import axios from 'axios';

export default () => dispatch => {
  dispatch({type: 'LOAD_TODOS'});

  return axios({
    method: 'get',
    url: 'localhost:3000'
  }).then(response => dispatch({type: 'COMPLETE_LOAD_TODOS', response});)
};
```

Требования:

1. В качестве клиента использовать приложение, разработанное в предыдущей работе
2. Реализовать сохранение задачи на сервере и загрузку задач с сервера

3. При выполнении запросов нужно показывать пользователю сообщение и блокировать интерфейс
4. Все взаимодействие с API должно происходить в actions с применением redux-thunk