Простое приложение на Riot.js

Онлайн пример приложения Riot Application

Исходный код

> Данное руководство распространяется совершенно бесплатно и без каких-либо ограничений. Авторство остаётся за сайтом riot-js.ru.

*Давайте разработаем простое приложение, которое охватывает некоторые основные аспекты одностраничных приложений. Мы пройдём вес цикл разработки и в конечном итоге, создадим небольшое приложение использующее REST API и реализующее все основные операции CRUD (Создание, Чтение, Обновление и Удаление). Для работы мы будем использовать Riot.js версии **3.13.2** (на момент написания руководства), Webpack последней версии и Node.js должен быть у вас установлен. Кроме этого, вы должны иметь базовые знания по **Riot.js**, поэтому, если их у вас нет, прочитайте Учебник и изучите API.*

Содержание:

Подготовительные работы

Мы начнём с создания рабочей директории и точки входа для нашего приложения. Создайте папку с названием **app**, а в ней создайте файл **index.html**:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Riot Application</title>
</head>
<body>
  <script src="dist/build.js"></script>
</body>
</html>
```

> Мы могли бы создать всё приложение в одном файле **Javascript**, но в дальнейшем, это затруднило бы навигацию по нашему коду. Вместо этого, давайте разделим код на модули и соберём эти модули в файл **dist/build.js**.

Перейдите в наш рабочий каталог **app**, затем откройте терминал командной строки, перейдите в эту папку в терминале и введите команду:

```
npm init -y
```

> Не забывайте в терминале переходить в папку **app**! Терминал должен ссылаться на неё при вводе всех команд.

Она создаст файл **package.json**, который содержит описание нашего проекта и управляет его зависимостями. Если вы всё сделали правильно, то вот так сейчас выглядит этот файл:

```
{
  "name": "app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Давайте заменим в этом файле содержимое секции **scripts**, и вместо:

```
"test": "echo \"Error: no test specified\" && exit 1"
```

пропишем:

```
"dev": "webpack-dev-server -d --open",
"build": "webpack -p"
```

Теперь наш файл должен выглядеть так:

```
{
  "name": "app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "webpack-dev-server -d --open",
    "build": "webpack -p"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Команда **dev** будет отвечать за запуск **Webpack** в режиме разработки, а команда **build** используется для продакшена. На этом, наши ручные манипуляции с файлом **package.json** можно считать законченными.

Для нашей задачи нам потребуется установить несколько пакетов. Сначала установим пакеты необходимые для разработки. Введите в терминале:

```
npm i -D webpack webpack-cli webpack-dev-server
```

Затем, установим пакеты для самого приложения:

```
npm i -S riot@3.13.2 riot-route
```

Таким образом, наш файл **package.json** получит следующее содержимое:

```
{
  "name": "app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "webpack-dev-server -d --open",
    "build": "webpack -p"
```

```
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^4.29.6",
    "webpack-cli": "^3.3.0",
    "webpack-dev-server": "^3.2.1"
  },
  "dependencies": {
    "riot": "^3.13.2",
    "riot-route": "^3.1.4"
  }
}
```

Создадим в нашей папке **app** подпапку **src**, а в неё добавим ещё три папки:

```
assets, models и views
```

Структура нашего проекта примет следующий вид:

```
app/
    node_modules/
    src/
        assets/
        models/
        views/
    index.html
    package.json
```

> Мы не создавали папку **node_modules**, она была создана **npm** (менеджер пакетов Node.js) автоматически, во время установки пакетов.

На этом, подготовительные работы для нашего проекта окончены. Следующим шагом будет создание файла конфигурации для **Webpack** и пробный запуск приложения.

Создание файла конфигурации Webpack

В нашей папке **app** создайте файл **webpack.config.js** и введите в него следующий код:

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

module.exports = {
  entry: './src/App.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'build.js',
    publicPath: 'dist/'
```

```
  }
}
```

Точкой входа приложения будет файл **App.js**, расположенный в папке **src**, а выводить наш код мы будем в файл **build.js**, расположеный в папке **dist**.

Создайте в папке **src** файл **App.js** и введите в него:

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

К этому моменту, структура нашего проекта имеет следующий вид:

```
app/
    node_modules/
    src/
        assets/
        models/
        views/
        App.js
    index.html
    package.json
    webpack.config.js
```

Сохраните файл и запустите терминал из папки **app**. В терминале введите команду:

```
npm run dev
```

После этого откроется страница в браузере, соответствующая файлу **index.html**. Перейдите в консоль браузера, там вы должны увидеть приветственное сообщение:

```
> Hello Riot!
```

Закройте файл **webpack.config.js** и перейдите в папку **views**, где мы создадим наш первый компонент **Hello**, который будет выводить приветственное сообщение на странице.

Создание компонента Hello

> Постарайтесь найти для своего редактора расширение, поддерживающие синтаксис **riot** в файлах **.tag**. Это сильно облегчит вам работу в дальнешей. Для редактора Visual Studio Code, такое дополнение имеется.

В папке **views** создайте файл **Hello.tag**. В данном файле будет распологаться наш компонент, который мы назовем **r-hello**.

> Все компоненты мы будем хранить в этой папке, поскольку они являются **представлениями** в терминологии **Riot.js**.

Префикс **r-** не является обязательным в названии тега компонента. Этим действием, мы лишь показываем его принадлежность к пользовательским тегам **riot** и, одновременно, избегаем пересечения в пространстве имён со стандартными **html-элементами**, наподобие **header**. Например, если бы нам потребовался компонент **header**, то мы назвали бы его **r-header**.

В файле **Hello.tag** введите:

```
<r-hello>
  <h1>Hello Riot!</h1>
</r-hello>
```

Затем, откройте файл **App.js**, удалите приветствие и подключите наш компонент к приложению:

```
// подключаем компонент Hello
import './views/Hello.tag'
```

Точка в начале названия пути **'./views/Hello.tag'** компонента, определяет относительный путь к нему от файла **App.js**.

> Нам **не требуется** импортировать наш компонент в переменную, вида:

```
import Hello from './views/Hello.tag'
```

> Поскольку файлы компонентов **не содержат** никакого экспорта, мы импортируем их содержимое прямо в файл нашего приложения.

Если мы сейчас попытаемся запустить **Webpack** командой:

```
npm run dev
```

то неизбежно получим сообщение об ошибке синтаксического разбора:

```
ERROR in ./src/views/Hello.tag 1:0
Module parse failed: Unexpected token (1:0)
You may need an appropriate loader to handle this file type.
> <r-hello>
|   <h1>Hello Riot!</h1>
| </r-hello>
 @ ./src/App.js 2:0-26
 @ multi (webpack)-dev-server/client?http://localhost:8080 ./src/App.js
```

Из которого можно сделать вывод, что нам потребуется установить соответствующий **загрузчик Webpack**.

> Несмотря на ошибку, **Webpack** не завершает своё выполнение в терминале. Остановить его работу можно командой **Ctrl+C**

Мы будем использовать загрузчик riot-tag-new-loader.

Откройте терминал из папки **app** или переведите терминал в неё другим способом, и введите команду:

```
npm i -D riot-tag-new-loader
```

> Флаг **-D** указывает на зависимость **devDependencies**, которая используется для процесса разработки, а флаг **-S** на **dependencies**, в которой указываются пакеты, используемые для работы самого приложения.

Наш файл **package.json** теперь выглядит так:

```
{
  "name": "app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "webpack-dev-server -d --open",
    "build": "webpack -p"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "riot-tag-new-loader": "^1.0.14",
    "webpack": "^4.29.6",
    "webpack-cli": "^3.3.0",
    "webpack-dev-server": "^3.2.1"
  },
  "dependencies": {
    "riot": "^3.13.2",
    "riot-route": "^3.1.4"
  }
}
```

Простая установка загрузчика, не избавит нас о вышеуказанной проблемы. Нам нужно будет добавить соответствующие **правила** в файле **webpack.config.js**.

Откроем файл **webpack.config.js** и добавим новое правило в массив **rules** объекта **module**:

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

module.exports = {
  entry: './src/App.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
```

```
    filename: 'build.js',
    publicPath: 'dist/'
  },
  module: {
    rules: [
      // добавляем новое правило для файлов компонентов (.tag)
      {
        test: /\.tag$/,
        exclude: /(node_modules|bower_components)/,
        use: 'riot-tag-new-loader'
      }
    ]
  }
}
```

Если мы сейчас снова попытаемся запустить **Webpack**:

```
npm run dev
```

то ошибка исчезнет, но ничего интересного, кроме пустой страницы в браузере, мы не увидим.

> Загрузчик лишь компилирует содержимое файлов компонентов в обычный **JavaScript**, который **Webpack** затем подключает к нашему приложению.

Нам потребуется выполнить три завершающих действия:

- подключить **Riot.js** к нашему приложению
- передать **Riot.js** тег **Hello** для монтирования
- подключить тег к странице **index.html**

Откроем файл **App.js** и в самом его верху, **перед** подключением компонента, добавим команду импорта:

```
import riot from 'riot'
```

а в конце файла, **после** подключением компонента, команду монтирования:

```
riot.mount('r-hello')
```

Теперь наш файл **App.js** должен выглядеть так:

```
// подключаем Riot.js
import riot from 'riot'

// подключаем компонент Hello
import './views/Hello.tag'
```

```
// монтируем компонент Hello
riot.mount('r-hello')
```

Мы передаём в **Riot.js** название компонента так, как мы указали его в файле **Hello.tag**, т.е. **r-hello**. Это же название мы будем использовать и при подключении компонента к странице, а **Hello.tag** - это просто название файла, в котором хранится наш компонент.

Откроем файл **index.html** и подключим компонент к странице:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Riot Application</title>
</head>
<body>

  <!-- подключаем компонент Hello к странице -->
  <r-hello />

  <script src="dist/build.js"></script>
</body>
</html>
```

И вот теперь, мы снова можем перезапустить **Webpack**:

```
npm run dev
```

после чего, откроется наша страница в браузере с приветственным сообщением:

Hello Riot!

Мы можем менять наше сообщение в компоненте **Hello**, и браузер автоматически будет обновлять страницу.

Это достигается благодаря тому, что мы в самом начале установили **webpack-dev-server**, а за само открытие страницы в браузере, отвечает его флаг **--open** команды **dev** в секции **scripts** файла **package.json**

```
"scripts": {
  "dev": "webpack-dev-server -d --open",
  "build": "webpack -p"
}
```

Мы закончили с ознакомительной частью данного руководства и, уже в следующей, перейдём непосредственно к написанию нашего приложения. И начнём мы с создания модуля для хранения его состояния.

Модуль хранения состояния

В папке **models** создайте файл **User.js**. Наше состояние будет храниться в экземпляре класса **User**. Давайте создадим этот класс, который будет иметь два свойства для хранения данных:

```
export default class User {

  constructor() {
    this.list = []
    this.current = {}
  }

}
```

- свойство **list** - это список пользователей
- свойство **current** - это текущий пользователь

> Заметьте, что мы используем **экспорт по умолчанию** для нашего класса. Это обычная практика при работе с модулями, содержащими один единственный класс для экспорта, что не отменяет одновременно и **именованный экспорт**, если в этом возникнет такая необходимость.

Теперь давайте добавим код для загрузки некоторых данных с сервера. Для связи с сервером мы будем использовать Fetch API, который является XMLHttpRequest нового поколения.

Создайте метод **getUsers**, который будет запускать вызов **XHR** и получать список пользователей с сервера:

```
export default class User {

  constructor() {
    this.list = []
    this.current = {}
  }

  // получаем список пользователей с сервера
  getUsers() {

  }

}
```

> В этом руководстве, мы будем делать вызовы **XHR** для REM API, который является фиктивным REST API для быстрого создания прототипов. Этот **API** возвращает список пользователей из конечной точки.

Используя **fetch**, выполним **XHR**-запрос и заполним наш список **list** данными из конечной точки:

```
export default class User {

  constructor() {
    this.list = []
    this.current = {}
  }

  // получаем список пользователей с сервера
  getUsers() {
    fetch('https://rem-rest-api.herokuapp.com/api/users', {
      method: 'GET',
      credentials: 'include'
    })
      .then(response => response.json())
      .then(result => {
        this.list = result.data
      })
  }

}
```

- свойство **method** - это метод HTTP
- свойство **credentials** - отвечает за Cookie

В первом аргументе **fetch** передаётся **url** адрес для конечной точки **API**, второй аргумент представляет собой объект с параметрами запроса. В свойстве **method** этого объекта пишем **GET**, а для свойства **credentials** задаём значение **include**, которое указывает на то, что мы используем **куки**, поскольку это является обязательным требованием для REM API.

Вызов **fetch** возвращает **промис**, который, когда ответ будет получен, выполняет функции обратного вызова с объектом Response или с ошибкой, если запрос не удался. Объект **response** предоставляет методы, позволяющие прочитать тело ответа в необходимом нам формате. В нашем случае, сервер возвращает нам ответ в формате **JSON**, который, с помощью метода **json()** объекта **response** в первом вызове **.then**, преобразуется в объект **JavaScript** и возвращается промис. Следующий **.then** присваивает полученные данные, которые представляют собой массив объектов **JavaScript**, свойству **list** нашего экземпляра класса **User**.

Теперь мы создадим компонет **UserList**, который является представлением и служит для отображения данных из нашего модуля состояния.

Создание компонента UserList

Из папки **views** удалите файл **Hello.tag**, поскольку компонент **Hello** нам больше не нужен. Удалите и его подключение из файлов **App.js** и **index.html**. Вот так теперь они должны у вас выглядеть:

App.js

```
// подключаем Riot.js
import riot from 'riot'
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Riot Application</title>
</head>
<body>
  <script src="build.js"></script>
</body>
</html>
```

В папке **views** создайте файл **UserList.tag**

```
<r-list>

  <div class="list">

  </div>

</r-list>
```

Наш компонент пока содержит только шаблон, в виде пустого списка с классом **list**. Стили и логику мы добавим позже. Теперь давайте обратимся к списку пользователей из модели, которую мы создали ранее, чтобы динамически пройти по всем данным.

Добавьте в компонент **UserList** ссылки на пользователей:

```
<r-list>

  <div class="list">

    <!-- добавляем ссылки на пользователей -->
    <a href="#!/edit/{ id }" class="list__item" each={ list } key={ id }>{ firstName
} { lastName }</a>

  </div>

</r-list>
```

В ссылке имеется атрибут **each**, который отвечает за реализацию циклов в **Riot.js**. Он получает свойство **list** объекта компонента, которое, впоследствии, будет ссылаться на одноимённое свойство экземпляра объекта класса **User**, и

представляет из себя массив объектов получаемых от сервера. Каждый объект в этом массиве содержит свойства: **id**, **firstName** и **lastName**. В приведённом выше примере, мы просто выводим содержимое этих свойст в ссылку для каждого пользователя.

> Выражения в **Riot.js** заключаются в одинарные фигурные скобки и могут располагаться как в атрибутах тегов, так и представлять их содержимое находясь между ними. Кавычки в атрибутах не являются обязательными.

Вот как выглядит JSON-файл получаемый с сервера:


```
{
  "data": [
    {
      "id": 1,
      "firstName": "Peter",
      "lastName": "Mackenzie"
    },
    {
      "id": 2,
      "firstName": "Cindy",
      "lastName": "Zhang"
    },
    {
      "id": 3,
      "firstName": "Ted",
      "lastName": "Smith"
    },
    {
      "id": 4,
      "firstName": "Susan",
      "lastName": "Fernbrook"
    },
    {
      "id": 5,
      "firstName": "Emily",
      "lastName": "Kim"
    },
    {
      "id": 6,
      "firstName": "Peter",
      "lastName": "Zhang"
    },
    {
      "id": 7,
      "firstName": "Cindy",
      "lastName": "Smith"
    },
    {
      "id": 8,
      "firstName": "Ted",
      "lastName": "Fernbrook"
    },
    {
      "id": 9,
      "firstName": "Susan",
```

```
      "lastName": "Kim"
    },
    {
      "id": 10,
      "firstName": "Emily",
      "lastName": "Mackenzie"
    }
  ],
  "offset": 0,
  "limit": 10,
  "total": 25
}
```

В объекте нашей модели данных класса **User**, мы определили свойство **list**, которое является массивом и будет хранить содержимое свойства **data**, возвращаемого с сервера файла **JSON**. Последние три свойства в этом файле: **offset**, **limit** и **total** используются исключительно для уточнения запросов. Мы воспользуемся свойством **limit** позже, для указания количества выводимых пользователей. По умолчанию оно равно 10.

Добавим немного стилей в наш компонент. Стили в компоненте размещаются между стандартными тегами **style**:

```
<r-list>

  <div class="list">

    <!-- добавляем ссылки на пользователей -->
    <a href="#!/edit/{ id }" class="list__item" each={ list } key={ id }>{ firstName
} { lastName }</a>

  </div>

  <!--  добавляем стили  -->
  <style>
    .list {
      list-style: none;
      margin: 0 0 10px;
      padding: 0;
    }
    .list__item {
      background: #fafafa;
      border: 1px solid #ddd;
      color: #333;
      display: block;
      margin: 0 0 1px;
      padding: 8px 15px;
      text-decoration: none;
    }
    .list__item:hover {
      text-decoration: underline;
    }
  </style>

</r-list>
```

> Файл компонента представляет из себя обычный **HTML**, но с некоторыми усовершенствованиями, засчет возможностей **Riot.js**.

Последнее, что нам осталось сделать, это определить логику нашего компонента. Её можно разместить как между тегами **script**, так и просто в теле компонента. Мы будем использовать первый вариант.

Итак, нам нужно получить список пользователей с сервера и вывести их в нашем компоненте. Т.е. нам нужно вызвать метод **getUsers** экземпляра класса **User**. Этот метод заполнит массив **list** данного класса полученными данными от сервера, а уже потом, мы сможем на него сослаться из нашего компонента и прогнать в цикле.

Для подобных целей, **Riot.js** предоставляет **примеси** и **наблюдателя**. Наблюдатель позволяет отслеживать события и выполнять соответствующие действия при их наступлении, а примеси расширяют функциональность нашего компонента, добавляя в него возможность эти события ловить и правильно на нах реагировать.

Давайте сделаем нашу модель данных наблюдаемой. Откройте файл **User.js** и в конструктор класса **User** добавьте параметр **riot**, через который мы будем ссылаться на библиотеку **Riot.js**, а в конце этого конструктора добавьте команду подключения наблюдателя:

```
riot.observable(this)
```

Эта команда делает **наблюдаемым**, каждый возвращаемый экземпляр объекта класса **User**:

```
constructor(riot) {
  this.list = []
  this.current = {}
  // делаем объект модели данных наблюдаемым
  riot.observable(this)
}
```

Теперь наш модуль хранения состояния выглядит так:

```
export default class User {

  constructor(riot) {
    this.list = []
    this.current = {}
    // делаем объект модели данных наблюдаемым
    riot.observable(this)
  }

  // получаем список пользователей с сервера
  getUsers() {
    fetch('https://rem-rest-api.herokuapp.com/api/users', {
```

```
      method: 'GET',
      credentials: 'include'
    })
      .then(response => response.json())
      .then(result => {
        this.list = result.data
      })
  }

}
```

Наш объект модели данных теперь сможет запускать события, но в этом не будет никакого смысла, поскольку обработчики этих событий должны располагаться в компонентах, а они, попросту пока не могут контактировать с нашим наблюдателем. И для этого, в **Riot.js** предусмотрены **примеси**, которые позволяют расширить функциональность наблюдателя нашей модели данных на любые компонеты.

Откройте главный файл приложения **App.js**. Давайте создадим примесь **user**, которая будет ссылаться на экземпляр наблюдаемой модели данных класса **User**:

```
// подключаем Riot.js
import riot from 'riot'

// подключаем модель данных User
import User from './models/User'

// создаём общую примесь user и передаём в конструктор модели данных User
// ссылку на библиотеку Riot.js, в виде аргумента riot
riot.mixin({ user: new User(riot) })
```

> Обратите внимание, что во время создания примеси **user**, мы передали в наш класс **ссылку** на библиотеку **Riot.js**, через аргумент **riot** класса **User**, которая будет доступна через одноимённый параметр данного класса. Также заметьте, что перед этим, мы подключили **модель данных** в приложение.

Осталось научить наши компоненты реагировать на события, которые будет генерировать наша модель данных, после успешного получения данных от сервера. Но прежде, давайте на минуту вернёмся в модуль нашей модели данных и научим нашу модель запускать эти самые события, после успешного ответа сервера.

Откройте файл **User.js** и в функции **getUsers**, в конце последнего **.then**, добавьте команду:

```
this.trigger('updated')
```

Она будет следовать сразу, после присвоения данных ответа сервера свойству **list**, нашей модели данных класса **User**:

```
.then(result => {
  // присваиваем результат ответа сервера свойству list модели данных
  this.list = result.data
  // запускаем событие updated, после успешного получения данных от сервера
  this.trigger('updated')
})
```

Полный код модуля хранения состояния должен выглядеть так:

```
export default class User {

  constructor(riot) {
    this.list = []
    this.current = {}
    // делаем объект модели данных наблюдаемым
    riot.observable(this)
  }

  // получаем список пользователей с сервера
  getUsers() {
    fetch('https://rem-rest-api.herokuapp.com/api/users', {
      method: 'GET',
      credentials: 'include'
    })
      .then(response => response.json())
      .then(result => {
        // присваиваем результат ответа сервера свойству list модели данных
        this.list = result.data
        // запускаем событие updated, после успешного получения данных от сервера
        this.trigger('updated')
      })
  }

}
```

Закончим с компонентом **UserList**. Откройте файл **UserList.tag**, добавьте в конце тела компонента теги **script**, и введите следующий код:

```
<script>

  // запускаем метод getUsers, нашей модели данных
  this.user.getUsers()

</script>
```

Поскольку мы расширили экземпляр модели данных класса **User** для любых компонентов с помощью примеси **user**, мы можем ссылаться на этот экземпляр данных, через одноимённое свойство самого **объекта компонента**, на которое указывает ключевое слово **this**.

Данный код будет выполнен до монтирования тега на страницу. Он запускает метод **getUsers** нашей модели данных, который, внутри себя, запускает **fetch**,

который выполняется асинхронно, и после загрузки данных с сервера, вызывает событие **updated**.

Теперь нам нужно отловить это событие модели данных в нашем компоненте и запустить событие **update** самого компонента:

```
<script>

  // запускаем метод getUsers, нашей модели данных
  this.user.getUsers()

  // запускаем событие обновления компонента (this.update)
  // при получении события updated от модели данных
  this.user.one('updated', this.update)

</script>
```

В котором, в свою очередь, мы присваиваем свойству **list** объекта компонента значение, одноимённого свойства **list** нашей модели данных:

```
<script>

  // запускаем метод getUsers, нашей модели данных
  this.user.getUsers()

  // запускаем событие обновления компонента (this.update)
  // при получении события updated от модели данных
  this.user.one('updated', this.update)

  // присваиваем свойству list нашего компонента значение
  // полученное моделью данных при запуске её метода getUsers()
  this.on('update', () => this.list = this.user.list)

</script>
```

после чего, компонет обновляется и полученные данные отображаются на странице.

Итоговый код компонента **UserList**:

```
<r-list>

  <div class="list">

    <!-- добавляем ссылки на пользователей -->
    <a href="#!/edit/{ id }" class="list__item" each={ list } key={ id }>{ firstName
} { lastName }</a>

  </div>

  <!--  добавляем стили  -->
  <style>
    .list {
```

```
      list-style: none;
      margin: 0 0 10px;
      padding: 0;
    }
    .list__item {
      background: #fafafa;
      border: 1px solid #ddd;
      color: #333;
      display: block;
      margin: 0 0 1px;
      padding: 8px 15px;
      text-decoration: none;
    }
    .list__item:hover {
      text-decoration: underline;
    }
  </style>

  <!--  добавляем логику  -->
  <script>

    // запускаем метод getUsers, нашей модели данных
    this.user.getUsers()

    // запускаем событие обновления компонента (this.update)
    // при получении события updated от модели данных
    this.user.one('updated', this.update)

    // присваиваем свойству list нашего компонента значение
    // полученное моделью данных при запуске её метода getUsers()
    this.on('update', () => this.list = this.user.list)

  </script>

</r-list>
```

Подключим компонент к приложению и примонтируем его в файле **App.js**:

```
// подключаем Riot.js
import riot from 'riot'

// подключаем модель данных User
import User from './models/User'

// подключаем компонент UserList
import './views/UserList.tag'

// создаём общую примесь user и передаём в конструктор модели данных User
// ссылку на библиотеку Riot.js, в виде аргумента riot
riot.mixin({ user: new User(riot) })

// монтируем компонент UserList
riot.mount('r-list')
```

а затем, подключим и к странице в файле **index.html**:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Riot Application</title>
</head>
<body>

  <!-- подключаем компонент UserList к странице -->
  <r-list />

  <script src="dist/build.js"></script>
</body>
</html>
```

Если мы теперь запустим из терминала команду:

```
npm run dev
```

то, откроется страница браузера со списком из 10 пользователей:

```
Peter Mackenzie
Cindy Zhang
Ted Smith
Susan Fernbrook
Emily Kim
Peter Zhang
Cindy Smith
Ted Fernbrook
Susan Kim
Emily Mackenzie
```

Мы увеличим лимит пользователей позже, когда будем реализовывать операции CRUD в другом компоненте, который мы тоже вскоре создадим.

Создание компонента App

Давайте изменим структуру нашего приложения и создадим входной компонент **App**. Данный компонент будет представлять точку входа нашего приложения для всех остальных компонентов.

В папке **views** создайте файл **App.tag**

```
<app>

</app>
```

Это будет единственный компонент, в котором мы не будем использовать префикс **r-**, а подключать к странице мы будем его немного другим способом, через атрибут **data-is** html-элемента **body**.

Откройте файл **index.html**, удалите подключение компонента **UserList** и добавьте атрибут **data-is** со значением **app**, к стандартному элементу **body**:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Riot Application</title>
</head>
<body data-is="app">
  <script src="dist/build.js"></script>
</body>
</html>
```

Теперь вернёмся к нашему компоненту **App** и подключим в него компонент **UserList**:

```
<app>

  <!--  подключаем компонент UserList  -->
  <r-list />

</app>
```

Последние, что нам осталось, это подключить наш компонент к приложению в файле **App.js** и передать его **Riot.js** для монтирования, вместо компонента **UserList**:

```
// подключаем компонент App
import './views/App.tag'

// монтируем компонент App
riot.mount('app')
```

А вот как должен выглядеть к этому моменту наш файл **App.js**:

```
// подключаем Riot.js
import riot from 'riot'

// подключаем модель данных User
import User from './models/User'

// подключаем компонент UserList
import './views/UserList.tag'
```

```
// подключаем компонент App
import './views/App.tag'

// создаём общую примесь user и передаём в конструктор модели данных User
// ссылку на библиотеку Riot.js, в виде аргумента riot
riot.mixin({ user: new User(riot) })

// монтируем компонент App
riot.mount('app')
```

И если мы теперь запустим из терминала команду:

```
npm run dev
```

то снова откроется страница браузера со списком из 10 пользователей:

```
Peter Mackenzie
Cindy Zhang
Ted Smith
Susan Fernbrook
Emily Kim
Peter Zhang
Cindy Smith
Ted Fernbrook
Susan Kim
Emily Mackenzie
```

В этом уроке мы создали компонент **App**, который является главным компонентом нашего приложения. В следующх уроках, мы создадим и добавим в него ещё несколько компонентов.

Создание компонента Menu

В папке **views** создайте файл **Menu.tag**

```
<r-menu>

  <a href="#!/list">Users</a>

  <style>
    :scope {
      margin-bottom: 10px;
    }
    a {
      padding: 5px;
      text-decoration: none;
      font-size: 22px;
    }
  </style>

</r-menu>
```

Это очень простое меню с одним единственным пунктом **Users**. Кроме этого, мы добавили нашему меню немного стилей.

> Обратите внимание, что в стилях мы используем псевдокласс **:scope**, который ссылается на сам компонент, т.е. на тег `<r-menu>`, который является контейнером для нашего компонента **Menu**.

Наш новый компонент **Menu** мы будем подключать к компоненту **Header**, который мы создадим далее.

Создание компонента Header

Давайте создадим шапку нашего приложения. В папке **views** создайте файл **Header.tag**

```
<r-header>

  <!--  подключаем компонент Menu  -->
  <nav data-is="r-menu" />

  <img src="dist/img/riot.png" alt="riot">

  <style>
    :scope {
      margin-top: 15px;
      margin-bottom: 30px;
      padding: 5px 15px;
    }
  </style>

</r-header>
```

В самом начале нашего компонента **Header**, мы сразу подключаем компонент **Menu**, который мы создали на предыдущем шаге, через атрибут **data-is** html-элемента **nav**.

> В данном случае, мы могли бы и не использовать атрибут **data-is** в html-элементе **nav**, а напрямую подключить наш компонет, как мы это делали ранее: `<r-menu />`. Но мы хотим следовать сементике и раскрыть некоторые возможности **Riot.js**, таким образом, всё содержимое нашего компонента **Menu**, окажется внутри html-элемента **nav**, который имеет атрибут **data-is**. Аналогичным образом мы поступим и с компонентом **Header**, и с компонентом **Footer**, который мы создадим далее, используя для этого соответствующие семантике - стандартные элементы html.

Ко всему прочему, мы также добавим в него немного стилей и в качестве содержимого, мы добавим в наш компонент картинку с официального сайта **Riot.js**.

Скачайте картинку: https://riot.js.org/img/logo/riot120x.png и переименуйте её в **riot.png**.

В нашем приложении имеется папка **assets**, которая нужна для размещения в ней всевозможных дополнительных файлов, необходимых нашему приложению. К ним можно отнести изображения, шрифты и т.д. Создайте в ней папку **img** и перенесите в неё изображение, которое мы скачали ранее.

Таким образом, структура нашего проекта, к данному моменту, должна иметь у вас следующий вид:

```
app/
    node_modules/
    src/
        assets/
          img/
            riot.png
        models/
          User.js
        views/
          App.tag
          Header.tag
          Menu.tag
          UserList.tag
        App.js
    index.html
    package.json
    webpack.config.js
```

Давайте подключим наш компонент **Header** к компоненту **App** и перейдём к созданию компонента **Footer**.

Откройте файл **App.tag** и добавьте в него компонент **Header** через атрибут **data-is**, стандартного html-элемента **header**:

```
<app>

  <!--  подключаем компонент Header  -->
  <header data-is="r-header" />

  <!--  подключаем компонент UserList  -->
  <r-list />

</app>
```

Мы пока не подключали наши новые компоненты к приложению в файле **App.js**. Мы сделаем это сразу, после создания компонента **Footer**, который будет представлять подвал нашего приложения.

Создание компонента Footer

Создадим простой подвал для нашего приложения, который будет содержать только ссылку на официальный сайт **Riot.js**.

В папке **views** создайте файл **Footer.tag**

```
<r-footer>

  <a href="https://riot.js.org/" target="_blank">Riot</a>

  <style>
    :scope {
      background: #222;
      margin-top: auto;
      padding: 25px 15px;
      text-align: center;
    }
    a {
      color: #fff;
    }
  </style>

</r-footer>
```

Теперь подключим наш компонент **Footer** к компоненту **App**, через атрибут **data-is**, стандартного html-элемента **footer**.

Откройте файл **App.tag** и добавьте в него компонент **Footer**:

```
<app>

  <!--  подключаем компонент Header  -->
  <header data-is="r-header" />

  <!--  подключаем компонент UserList  -->
  <r-list />

  <!--  подключаем компонент Footer  -->
  <footer data-is="r-footer" />

</app>
```

Последним шагом, мы подключим все три компонента к нашему приложению в файле **App.js**:

```
// подключаем компонент Menu
import './views/Menu.tag'

// подключаем компонент Header
import './views/Header.tag'

// подключаем компонент Footer
import './views/Footer.tag'
```

Таким образом, наш файл **App.js** теперь имеет следующий вид:

```
// подключаем Riot.js
import riot from 'riot'

// подключаем модель данных User
import User from './models/User'

// подключаем компонент UserList
import './views/UserList.tag'

// подключаем компонент App
import './views/App.tag'

// подключаем компонент Menu
import './views/Menu.tag'

// подключаем компонент Header
import './views/Header.tag'

// подключаем компонент Footer
import './views/Footer.tag'

// создаём общую примесь user и передаём в конструктор модели данных User
// ссылку на библиотеку Riot.js, в виде аргумента riot
riot.mixin({ user: new User(riot) })

// монтируем компонент App
riot.mount('app')
```

Если мы сейчас снова запустим наше приложение командой:

```
npm run dev
```

то откроется страница в браузере, которая будет вверху иметь меню, состоящее из одного пункта **Users**, хедер, содержащий неработающую картинку и футер, с сылкой на официальный сайт **Riot.js**.

Кроме этого, поскольку мы никак не сбрасывали стили, наши компоненты будут иметь некоторые отступы по бокам. Исправим это, добавив нормализацию стилей в главный и единственный html-файл нашего приложения **index.html**:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Riot Application</title>
  <!-- подключаем normalize.css -->
  <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.1/normalize.min.css"/>
</head>
<body data-is="app">
  <script src="dist/build.js"></script>
```

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

Ко всему прочему, мы бы ещё хотели, чтобы наш футер прижимался к нижней части экрана, когда содержимое нашего списка слишком мало и под ним остаётся свободное место. И для этого, мы добавим немного стилей к компоненту **App**, в которых задействуем флексы:

```
<app>

  <!--  подключаем компонент Header  -->
  <header data-is="r-header" />

  <!--  подключаем компонент UserList  -->
  <!-- <r-list /> -->

  <!--  подключаем компонент Footer  -->
  <footer data-is="r-footer" />

  <!-- добавляем стили -->
  <style>
    :scope {
      display: flex;
      flex-direction: column;
      font: normal 16px Verdana;
    }
  </style>

</app>
```

Но сейчас это не сработает, пока мы явно не зададим высоту для нашего приложения и документа, в котором оно содержится, равной ста процентам высоты экрана устройства, на котором оно будет отображаться.

> Мы не можем поместить стили для документа, в частности, для элемента **html** в компонент **App** нашего приложения, поскольку в css невозможно получить доступ к родительским элементам из дочерних. Поэтому, мы могли бы создать отдельный css-файл, в котором располагались бы стили, переопределяющие или дополняющие стили нашего приложения. Так мы и поступим позднее, а пока, мы временно добавим их в файл **index.html**, сразу после подключения **normalize.css**.

Откройте файл **index.html** и добавьте стили для документа нашего приложения:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Riot Application</title>
  <!-- подключаем normalize.css -->
  <link rel="stylesheet"
```

```
href="https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.1/normalize.min.css"/>
  <!-- стили документа -->
  <style>
    html, body {
      height: 100%;
    }
  </style>
</head>
<body data-is="app">
  <script src="dist/build.js"></script>
</body>
</html>
```

Теперь, если мы запустим наше приложение командой:

```
npm run dev
```

и откроем его в браузере, то футер будет прижиматься к нижней части экрана, когда содержимое списка слишком мало, чтобы полностью в нём уместиться. Но картинка по-прежнему отображаться не будет, и для того, чтобы это исправить, нам нужно вернуться в конфигурационный файл **webpack.config.js**.

Завершение файла конфигурации Webpack

Пришло время закончить с нашим файлом конфигурации. В этом уроке, мы выполним все необходимые действия для успешной работы нашего приложения, а в последующих, напишем ещё один компонент и добавим маршрутизацию к нашему приложению.

Сейчас наш файл конфигурации **webpack.config.js**, должен выглядеть вот так:

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

module.exports = {
  entry: './src/App.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'build.js',
    publicPath: 'dist/'
  },
  module: {
    rules: [
      // добавляем новое правило для файлов компонентов (.tag)
      {
        test: /\.tag$/,
        exclude: /(node_modules|bower_components)/,
        use: 'riot-tag-new-loader'
      }
    ]
  }
}
```

Первым делом, научим **Webpack** копировать файлы. И для этого, мы будем использовать плагин copy-webpack-plugin.

Давайте его установим. Откройте терминал из папки **app** нашего приложения и введите команду:

```
npm i -D copy-webpack-plugin
```

В самом верху файла **webpack.config.js**, после подключения модуля путей **path**, добавим строку подключения нашего плагина:

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

// подключаем плагин для копирования файлов
const CopyPlugin = require('copy-webpack-plugin')
```

Далее, в этом же файле, добавим в объект конфигурации новое свойство **plugins**, сразу после свойства **module**. И в нём зарегистрируем наш плагин и пропишем ему настройки для копирования файлов:

```
plugins: [
  // регистрируем плагин copy-webpack-plugin и задаём ему необходимые параметры
  new CopyPlugin([
    {
      from: 'src/assets/**/*.{png,jpg}',
      to: 'img',
      flatten: true
    }
  ])
]
```

- **from** - откуда и какие файлы копировать
- **to** - в какую папку копировать
- **flatten** - удалять ссылки на каталоги и копировать только файлы

После всех манипуляций, вот так должен выглядть файл **webpack.config.js**:

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

// подключаем плагин для копирования файлов
const CopyPlugin = require('copy-webpack-plugin')

module.exports = {
  entry: './src/App.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'build.js',
    publicPath: 'dist/'
  },
  module: {
    rules: [
```

```
      // добавляем новое правило для файлов компонентов (.tag)
      {
        test: /\.tag$/,
        exclude: /(node_modules|bower_components)/,
        use: 'riot-tag-new-loader'
      }
    ]
  },
  plugins: [
    // регистрируем плагин copy-webpack-plugin и задаём ему необходимые параметры
    new CopyPlugin([
      {
        from: 'src/assets/**/*.{png,jpg}',
        to: 'img',
        flatten: true
      }
    ])
  ]
}
```

Таким образом, все наши изображения будут помещены в папку **img**, которая будет располагаться в папке **dist**.

Если мы сейчас запустим наше приложение:

```
npm run dev
```

то откроется страница, в верху которой будет отображаться меню с одним единственным пунктом **Users**, а под ним появится наша картинка-логотип, которую мы прежде скачали, переименовали и поместили в папку **assets**.

> Вы могли заметить, что у нас до сих пор не появилось никакой папки **dist** в каталоге **app** нашего приложения. Это связано с тем, что мы запускаем **Webpack** в режиме разработки `npm run dev`, а не в режиме продакшена `npm run build`. В режиме разработки, **Webpack**, условно, создаёт её виртуальную копию, путь к которой прописан в свойстве **publicPath** свойства **output**, объекта конфигурации **Webpack**:

```
output: {
  path: path.resolve(__dirname, 'dist'),
  filename: 'build.js',
  publicPath: 'dist/'
}
```

Следующее, что мы сделаем, мы установим и подключим **Babel**, который будет преобразовывать всю нашу **ES6** логику из компонентов и переводить её в код **ES5** для браузеров, которые плохо понимают современные стандарты **JavaScript**. И для этого, нам потребуется babel-loader.

Откройте терминал и введите команду:

```
npm install -D babel-loader @babel/core @babel/preset-env
```

Нам также потребуется немного изменить наше правило для файлов компонентов:

```
module: {
  rules: [
    // добавляем новое правило для файлов компонентов (.tag)
    {
      test: /\.tag$/,
      exclude: /(node_modules|bower_components)/,
      use: [
        {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        },
        {
          loader: 'riot-tag-new-loader'
        }
      ]
    }
  ]
}
```

> Правила **Webpack** работают справа налево и снизу вверх. Т.е. сначала, загрузчик **riot-tag-new-loader** получает наши компоненты из файлов **.tag** и преобразует их в обычный **JavaScript**, а уже затем, загрузчик **babel-loader** преобразует этот код в **ES5**.

Мы бы ещё хотели, чтобы **css**, который мы пишем в компонентах, имел бы на выходе префиксы производителей для старых браузеров.

Откройте терминал и введите команду:

```
npm i -D postcss autoprefixer
```

Подключим эти модули в нашем файле конфигурации, после подключения плагина для копирования файлов:

```
// подключаем плагин для копирования файлов
const CopyPlugin = require('copy-webpack-plugin')

// подключаем модули для обработки css в компонентах
const postcss = require('postcss')
const autoprefixer = require('autoprefixer')
```

Вернёмся к правилу для файлов компонентов и добавим в него оъект параметров **riot-tag-new-loader**, а в нём создадим новый пользовательский

парсер css, которому передадим установленные выше модули:

```
module: {
  rules: [
    // добавляем новое правило для файлов компонентов (.tag)
    {
      test: /\.tag$/,
      exclude: /(node_modules|bower_components)/,
      use: [
        {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        },
        {
          loader: 'riot-tag-new-loader',
          // объект параметров riot-tag-new-loader
          options: {
            parsers: {
              // создаём пользовательский парсер css и передаём ему модули
              // для добавления префиксов к стилям наших компонентов
              css: {
                plain: function(tag, css) {
                  return postcss([ autoprefixer({ browsers: ['last 15 versions'] }) 
]).process(css).css
                }
              }
            }
          }
        }
      ]
    }
  ]
}
```

Мы назвали наш парсер **plain**, но могли бы называть как угодно. Данное название, как нельзя лучше, описывает тип стилей наших компонентов - это **простой** css. Для того, чтобы наш парсер смог обрабатывать эти стили, нам необходимо будет задать им тип, соответствующий названию нашего парсера, в тегах **style** каждого компонента.

Пройдитесь по всем компонентам содержащим стили, и добавьте в их теги **style** атрибут **type** со значением **plain**:

```
<style type="plain">
    /* стили компонента */
</style>
```

Наш главный файл приложения **App.js** может содержать в себе код **ES6**, кроме этого, вы можете подключать к нему другие файлы **JavaScript**, которые тоже будут содержать в себе код, отвечающий последним стандарнам **JS**. И для того,

чтобы все браузеры смогли с ним работать, эти файлы нам тоже придётся пропустить через **Babel**.

Добавим новое правило для файлов **JavaScript** в массив **rules** объекта конфигурации **Webpack**:

```
// добавляем новое правило для файлов JavaScript
{
  test: /\.js$/,
  exclude: /(node_modules|bower_components)/,
  use: {
    loader: 'babel-loader',
    options: {
      presets: ['@babel/preset-env']
    }
  }
}
```

К этому моменту, файл **webpack.config.js** должен выглядеть так:

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

// подключаем плагин для копирования файлов
const CopyPlugin = require('copy-webpack-plugin')

// подключаем модули для обработки css в компонентах
const postcss = require('postcss')
const autoprefixer = require('autoprefixer')

module.exports = {
  entry: './src/App.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'build.js',
    publicPath: 'dist/'
  },
  module: {
    rules: [
      // добавляем новое правило для файлов компонентов (.tag)
      {
        test: /\.tag$/,
        exclude: /(node_modules|bower_components)/,
        use: [
          {
            loader: 'babel-loader',
            options: {
              presets: ['@babel/preset-env']
            }
          },
          {
            loader: 'riot-tag-new-loader',
            // объект параметров riot-tag-new-loader
            options: {
              parsers: {
                // создаём пользовательский парсер css и передаём ему модули
```

```
                // для добавления префиксов к стилям наших компонентов
                css: {
                  plain: function(tag, css) {
                    return postcss([ autoprefixer({ browsers: ['last 15 versions'] }) 
]).process(css).css
                  }
                }
              }
            }
          }
        ]
      },
      // добавляем новое правило для файлов JavaScript
      {
        test: /\.js$/,
        exclude: /(node_modules|bower_components)/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        }
      }
    ]
  },
  plugins: [
    // регистрируем плагин copy-webpack-plugin и задаём ему необходимые параметры
    new CopyPlugin([
      {
        from: 'src/assets/**/*.{png,jpg}',
        to: 'img',
        flatten: true
      }
    ])
  ]
}
```

Нам осталось создать папку для стилей, которые не могут или не должны располагаться в файлах компонентов, на примере того, как мы перед этим вынесли стили для документа в файл **index.html**.

В папке **src** создайте папку **sass**, а в ней создайте файл **styles.scss**.

Структура проекта примет следующий вид:

```
app/
    node_modules/
    src/
        assets/
          img/
            riot.png
        models/
          User.js
        sass/
          styles.scss
```

```
        views/
          App.tag
          Footer.tag
          Header.tag
          Menu.tag
          UserList.tag
        App.js
    index.html
    package.json
    webpack.config.js
```

Откройте файл **styles.scss** и добавьте в него стили для документа:

```
// стили документа
html, body {
  height: 100%;
}
```

Теперь откройте файл **index.html** и удалите из него теги **style** вместе со стилями для документа, поскольку, мы вынесли их в отдельный файл **styles.scss**.

> Если вы ещё не познакомились с Sass, который является препроцессором **css**, то просто пишите в файле **styles.scss** свои стили так, как вы пишете их в обычном файле **css**.

Кроме этого, мы добавим подключение в **index.html** внешнего файла **build.css**, в котором и будут храниться все стили, которые мы пишем в файле **styles.scss**.

> Препроцессор **Sass** на выходе возвращает обычный файл **css**.

Но перед подключением файла **build.css**, мы укажем **Riot.js**, куда он должен будет помещать стили из компонентов в тег **head** файла **index.html**. Они должны быть помещены перед стилями из внешнего файла **build.css**, поскольку, предполагается, что стили в этом файле должны дополнять или переопределять стили компонентов. Согласно каскадности в **css**, последние добавленные стили с одинаковым приоритетом, переопределяют стили добавленные первыми. Для того, чтобы **Riot.js** понимал, куда ему нужно добавлять стили из компонентов, используется пустой тег **style** с атрибутом **type** и значением **riot**.

Таким образом, наш файл **index.html** теперь должен выглядеть так:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Riot Application</title>
  <!-- подключаем normalize.css -->
  <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.1/normalize.min.css"/>
```

```
  <!-- стили компонентов -->
  <style type="riot"></style>
  <!-- внешние стили -->
  <link rel="stylesheet" href="dist/build.css">
</head>
<body data-is="app">
  <script src="dist/build.js"></script>
</body>
</html>
```

Стили в **Webpack** подключаются к **JavaScript**, из которого, они потом достаются и обрабатываются специальными загрузчиками. Откройте файл **App.js** и после подключения компонента **Footer**, добавьте подключение внешнего файла стилей:

```
// подключаем компонент Footer
import './views/Footer.tag'

// подключаем внешние стили
import './sass/styles.scss'
```

Как было сказано выше, стили подключатся к **JavaScript**, из которого они потом обрабатываются специальными загрузчиками **Webpack**. Запустите терминал из папки **app** нашего приложения и введите команду:

```
npm i -D css-loader style-loader postcss-loader sass-loader node-sass
```

Добавим новое правило для **внешних стилей**, сразу после правила для файлов **JavaScript**:

```
// добавляем новое правило для внешних стилей (.scss)
{
  test: /\.scss$/,
  use: [
    'style-loader',
    'css-loader',
    'postcss-loader',
    'sass-loader'
  ]
}
```

Кроме этого, в папке **app** нашего приложения, там, где у нас находится файл **webpack.config.js**, создайте файл конфигурации **postcss.config.js** для **postcss-loader** и добавьте в него:

```
module.exports = {
  plugins: {
    autoprefixer: {
      browsers: ['last 15 versions']
    }
```

```
  }
}
```

В этом файле, мы просто указывает плагины, которые будет использовать **postcss-loader**. У нас таких плагинов всего один, это **autoprefixer**, который мы установили ранее.

Структура проекта должна принять вид:

```
app/
    node_modules/
    src/
        assets/
          img/
            riot.png
        models/
          User.js
        sass/
          styles.scss
        views/
          App.tag
          Footer.tag
          Header.tag
          Menu.tag
          UserList.tag
        App.js
    index.html
    package.json
    postcss.config.js
    webpack.config.js
```

а содержимое файла конфигурации **webpack.config.js**, должно быть таким:

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

// подключаем плагин для копирования файлов
const CopyPlugin = require('copy-webpack-plugin')

// подключаем модули для обработки css в компонентах
const postcss = require('postcss')
const autoprefixer = require('autoprefixer')

module.exports = {
  entry: './src/App.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'build.js',
    publicPath: 'dist/'
  },
  module: {
    rules: [
      // добавляем новое правило для файлов компонентов (.tag)
      {
        test: /\.tag$/,
```

```
        exclude: /(node_modules|bower_components)/,
        use: [
          {
            loader: 'babel-loader',
            options: {
              presets: ['@babel/preset-env']
            }
          },
          {
            loader: 'riot-tag-new-loader',
            // объект параметров riot-tag-new-loader
            options: {
              parsers: {
                // создаём пользовательский парсер css и передаём ему модули
                // для добавления префиксов к стилям наших компонентов
                css: {
                  plain: function(tag, css) {
                    return postcss([ autoprefixer({ browsers: ['last 15 versions'] }) 
]).process(css).css
                  }
                }
              }
            }
          }
        ]
      },
      // добавляем новое правило для файлов JavaScript
      {
        test: /\.js$/,
        exclude: /(node_modules|bower_components)/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        }
      },
      // добавляем новое правило для внешних стилей (.scss)
      {
        test: /\.scss$/,
        use: [
          'style-loader',
          'css-loader',
          'postcss-loader',
          'sass-loader'
        ]
      }
    ]
  },
  plugins: [
    // регистрируем плагин copy-webpack-plugin и задаём ему необходимые параметры
    new CopyPlugin([
      {
        from: 'src/assets/**/*.{png,jpg}',
        to: 'img',
        flatten: true
      }
    ])
```

```
  ]
}
```

Как мы помним, правила работают справа налево и снизу вверх. Т.е. сначала, **sass-loader** преобразует стили хранящиеся в файле **styles.scss** и написанные по правилам препроцессора **Sass** в обычный **css**, затем, этот **css** поступает в **postcss-loader**, в конфигурационном файле которого указан плагин **autoprefixer**, после этого, стили с добавленными префиксами производителей браузеров передаются в **css-loader**, который разрешает пути в **css**, а затем, они поступают в **style-loader**, который сохраняет их в **JavaScript** и подгружает в тег **head** нашего файла **index.html**.

Если мы сейчас запустим наш проект в режиме продакшена:

```
npm run build
```

то в папке **app** нашего приложения появится каталог **dist**, в котором будут находиться файл **build.js** и папка **img** с картинкой-логотипом **riot.png**. При этом, страница приложения не будет открыта автоматически в браузере, а **Webpack** завершит своё выполнение в командной строке терминала. После этого, мы можем передать файл **index.html** и папку **dist**, потенциальному заказчику нашего экспериментального приложения.

Но мы бы хотели, чтобы внешние стили у нас были всё-таки в отдельном файле **css**, а не в коде **JavaScript** файла **build.js**. И для этого, установим плагин mini-css-extract-plugin:

```
npm i -D mini-css-extract-plugin
```

Откроем файл **webpack.config.js** и добавим подключение этого плагина в самом верху, сразу после подключения модулей для обработки **css** в компонентах:

```
// подключаем модули для обработки css в компонентах
const postcss = require('postcss')
const autoprefixer = require('autoprefixer')

// подключаем плагин для извлечения css в отдельный файл
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
```

И сразу зарегистрируем его в нижней части файла, в секции **plugins**:

```
plugins: [
  // регистрируем плагин copy-webpack-plugin и задаём ему необходимые параметры
  new CopyPlugin([
    {
      from: 'src/assets/**/*.{png,jpg}',
      to: 'img',
      flatten: true
```

```
    }
  ]),
  // регистрируем плагин mini-css-extract-plugin и задаём ему необходимые параметры
  new MiniCssExtractPlugin({
    filename: 'build.css'
  })
]
```

Мы передаём плагину всего один параметр, это имя нашего итогового файла **css**. Как мы помним, он будет называться **build.css**.

Для режима разработки, мы будем использовать загрузчик стилей **style-loader**, а в режиме продакшена задействуем плагин **MiniCssExtractPlugin**.

Давайте внесём небольшие изменения в правило для внешних стилей:

```
// добавляем новое правило для внешних стилей (.scss)
{
  test: /\.scss$/,
  use: [
    // для продакшена используется плагин MiniCssExtractPlugin,
    // а для разработки будет применяться загрузчик style-loader
    options.mode === 'production' ? MiniCssExtractPlugin.loader : 'style-loader',
    'css-loader',
    'postcss-loader',
    'sass-loader'
  ]
}
```

Но откуда взялось **options.mode**?

Если мы внимательно посмотрим, то наш файл конфигурации **webpack.config.js** - это просто модуль **Node.js**, который экспортирует объект конфигурации **Webpack**:

```
module.exports = {
  // содержит точку входа, вывода, правила и плагины
}
```

Можно экспортировать не только объект, но и функцию, которая будет иметь некоторые параметры, которыми мы сможем воспользоваться для получения нужной нам информации. Другими словами, мы можем сделать так:

```
module.exports = (env, options) => {
  return {
    // содержит точку входа, вывода, правила и плагины
  }
}
```

Второй параметр этой функции, т.е. параметр **options**, содержит свойство **mode**, которое позволяет определить, в каком режиме был запущен **Webpack**.

Давайте изменим наш файл конфигурации в последний раз и вместо экспорта объекта конфигурации, мы экспортируем функцию с двумя параметрами, которая, в свою очередь, будет возвращать этот самый объект конфигурации **Webpack**.

Откройте файл **webpack.config.js** и внесите в него изменения, в соответствии с приведённым выше примером.

Финальный вид нашего файла конфигурации:

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

// подключаем плагин для копирования файлов
const CopyPlugin = require('copy-webpack-plugin')

// подключаем модули для обработки css в компонентах
const postcss = require('postcss')
const autoprefixer = require('autoprefixer')

// подключаем плагин для извлечения css в отдельный файл
const MiniCssExtractPlugin = require('mini-css-extract-plugin')

// экспортируем функцию с двумя параметрами
module.exports = (env, options) => {

  // функция возвращает объект конфигурации Webpack
  return {
    entry: './src/App.js',
    output: {
      path: path.resolve(__dirname, 'dist'),
      filename: 'build.js',
      publicPath: 'dist/'
    },
    module: {
      rules: [
        // добавляем новое правило для файлов компонентов (.tag)
        {
          test: /\.tag$/,
          exclude: /(node_modules|bower_components)/,
          use: [
            {
              loader: 'babel-loader',
              options: {
                presets: ['@babel/preset-env']
              }
            },
            {
              loader: 'riot-tag-new-loader',
              // объект параметров riot-tag-new-loader
              options: {
                parsers: {
                  // создаём пользовательский парсер css и передаём ему модули
                  // для добавления префиксов к стилям наших компонентов
                  css: {
                    plain: function(tag, css) {
                      return postcss([ autoprefixer({ browsers: ['last 15 versions']
```

```
}) ]).process(css).css
                    }
                  }
                }
              }
            }
          ]
        },
        // добавляем новое правило для файлов JavaScript
        {
          test: /\.js$/,
          exclude: /(node_modules|bower_components)/,
          use: {
            loader: 'babel-loader',
            options: {
              presets: ['@babel/preset-env']
            }
          }
        },
        // добавляем новое правило для внешних стилей (.scss)
        {
          test: /\.scss$/,
          use: [
            // для продакшена используется плагин MiniCssExtractPlugin,
            // а для разработки будет применяться загрузчик style-loader
            options.mode === 'production' ? MiniCssExtractPlugin.loader : 'style-
loader',
            'css-loader',
            'postcss-loader',
            'sass-loader'
          ]
        }
      ]
    },
    plugins: [
      // регистрируем плагин copy-webpack-plugin и задаём ему необходимые параметры
      new CopyPlugin([
        {
          from: 'src/assets/**/*.{png,jpg}',
          to: 'img',
          flatten: true
        }
      ]),
      // регистрируем плагин mini-css-extract-plugin и задаём ему необходимые
параметры
      new MiniCssExtractPlugin({
        filename: 'build.css'
      })
    ]
  }

}
```

Если снова запустить наш проект в режиме продакшена:

```
npm run build
```

то в папке **dist** окажется файл стилей **build.css**.

> Папка **dist** создаётся в режиме продакшена, когда вы в первый раз в нём запускаете **Webpack**. Для режима разработки эта папка не нужна.

Мы закончили с нашим файлом конфигурации **Webpack**. Теперь нам осталось создать последний компонент для работы с пользователями и добавить маршрутизацию к нашему приложению.

Создание компонента UserForm

Это будет наш последний компонент в приложении. Он будет содержать в себе форму с двумя полями ввода и три кнопки. Данный компонент реализует все операции CRUD (Создание, Чтение, Обновление и Удаление) для пользователей из списка.

В папке **views** создайте файл **UserForm.tag**

```
<r-form>

  <form class="form">
    <label class="form__label">First name</label>
    <input type="text" class="form__input" oninput={ inputValue } data-
name="firstName" placeholder="First name" value="{ firstName }">
    <label class="form__label">Last name</label>
    <input type="text" class="form__input" oninput={ inputValue } data-
name="lastName" placeholder="Last name" value="{ lastName }">
    <button class="form__btn form__btn--create" onclick={ clickButton } data-
method="createUser">Create</button>
    <button class="form__btn form__btn--delete" onclick={ clickButton } data-
method="deleteUser">Delete</button>
    <button class="form__btn form__btn--update" onclick={ clickButton } data-
method="updateUser">Update</button>
  </form>

  <style type="plain">
    .form__label {
      display: inline-block;
      margin: 0 0 5px;
    }
    .form__input {
      border: 1px solid #ddd;
      border-radius: 3px;
      box-sizing: border-box;
      display: block;
      margin: 0 0 10px;
      padding: 10px 15px;
      width: 100%;
    }
    .form__btn {
      background: #ddd;
      border: 1px solid #ddd;
      border-radius: 3px;
```

```
      color: #fff;
      cursor: pointer;
      display: inline-block;
      margin-top: 20px;
      padding: 10px 15px;
      text-decoration: none;
    }
    .form__btn--create {
      background: #41BA5E;
    }
    .form__btn--delete {
      background: #FF0044;
    }
    .form__btn--update {
      background: #0B77B3;
    }
    .form__btn:hover {
      background: #ddd;
      color: #222;
    }
  </style>

  <script>
    // обработчик события ввода в текстовое поле
    inputValue(e) {
      e.preventUpdate = true
      this.user.current[e.target.dataset.name] = e.target.value
    }

    // обработчик события нажатия на кнопку
    clickButton(e) {
      e.preventDefault()
      e.preventUpdate = true
      this.user[e.target.dataset.method]()
    }

    // переходим на страницу list (список пользователей)
    // при получении события home от модели данных
    this.user.one('home', () => this.route('list'))

    // обработчик события маршрутизатора
    this.on('route', (id) => this.user.getUser(id))

    // запускаем событие обновления компонента (this.update)
    // при получении события updated от модели данных
    this.user.one('updated', this.update)

    // обработчик события обновления компонента
    this.on('update', () => {
      this.firstName = this.user.current.firstName
      this.lastName = this.user.current.lastName
    })
  </script>

</r-form>
```

Структура компонента к этому моменту, должна быть вам уже хорошо знакома. Не забываем только в теге стилей компонентов **style**, указывать тип **plain**.

Давайте сначала подключим наш компонент к приложению в файле **App.js**, а потом вернёмся и рассмотрим его шаблон и логику более подробно.

Откройте файл **App.js** и перед подключением внешних стилей, но после подключения компонента **Footer**, добавим подключение компонента **UserForm**:

```
// подключаем компонент Footer
import './views/Footer.tag'

// подключаем компонент UserForm
import './views/UserForm.tag'

// подключаем внешние стили
import './sass/styles.scss'
```

> Мы могли бы подключить компонент и после подключения внешних стилей. Так просто выглядит более логично и соответствует порядку создания наших компонентов.

Файл **App.js** нам пока больше не нужен. Вернёмся к нашему компоненту **UserForm** и рассмотрим его шаблон:

```
<form class="form">
  <label class="form__label">First name</label>
  <input type="text" class="form__input" oninput={ inputValue } data-name="firstName"
placeholder="First name" value="{ firstName }">
  <label class="form__label">Last name</label>
  <input type="text" class="form__input" oninput={ inputValue } data-name="lastName"
placeholder="Last name" value="{ lastName }">
  <button class="form__btn form__btn--create" onclick={ clickButton } data-
method="createUser">Create</button>
  <button class="form__btn form__btn--delete" onclick={ clickButton } data-
method="deleteUser">Delete</button>
  <button class="form__btn form__btn--update" onclick={ clickButton } data-
method="updateUser">Update</button>
</form>
```

Как мы помним, выражения в **Riot.js** указываются между двумя фигурными скобками и могут располагаться как в атрибутах html-элементов, так и представлять их содержимое, т.е. находиться между их тегами. Кавычки в атрибутах для выражений не имеют значения.

Форма нашего компонента содержит два текстовых поля ввода и три кнопки. Оба поля имеют пользовательские атрибуты **data-name**, первое поле в нём содержит значение **firstName**, а второе, значение **lastName**:

```
<form class="form">
  <label class="form__label">First name</label>
  <input type="text" class="form__input" oninput={ inputValue } data-name="firstName"
```

```
placeholder="First name" value="{ firstName }">
  <label class="form__label">Last name</label>
  <input type="text" class="form__input" oninput={ inputValue } data-name="lastName"
placeholder="Last name" value="{ lastName }">
</form>
```

Кроме этого, оба поля вызывают функцию **inputValue**, при наступлении события **input**. Для этого, в значении атрибута **oninput** каждого из полей, содержится ссылка на эту самую функцию:

```
oninput={ inputValue }
```

Рассмотрим эту функцию:

```
// обработчик события ввода в текстовое поле
inputValue(e) {
  e.preventUpdate = true
  this.user.current[e.target.dataset.name] = e.target.value
}
```

Она начинается с прерывания события обновления компонента:

```
e.preventUpdate = true
```

Событие обновления компонента (**this.update**) вызывается всякий раз, когда вызывается какая-либо пользовательская функция, но в данном случае, нет никакой необходимости обновлять компонент и тратить на это время и ресурсы браузера. Если бы мы этого не сделали, то после каждого ввода символа в любое из текстовых полей, вызывалось бы это событие и запускался бы его обработчик:

```
// обработчик события обновления компонента
this.on('update', () => {
  this.firstName = this.user.current.firstName
  this.lastName = this.user.current.lastName
})
```

Второй командой в функции **inputValue**, является:

```
this.user.current[e.target.dataset.name] = e.target.value
```

Эта команда обращается к объекту **current** нашей модели данных **User**, и присваивает его свойству значение из соответствующего поля ввода, каждый раз, когда в это поле вводятся какие-то данные. Название свойства для объекта **current** берётся из атрибута **data-name** текстового поля, с которым в данный момент осуществляется интерактивное взаимодействие.

Как мы помним, наша модель данных **User** является общедоступной для всех компонентов благодаря тому, что мы вынесли её в общую примесь **user** в файле **App.js**:

```
riot.mixin({ user: new User(riot) })
```

В функции **inputValue**, компонент ссылается на примесь **user**, через одноимённое свойство:

```
this.user
```

Теперь рассмотрим кнопки в нашем шаблоне:

```
<button class="form__btn form__btn--create" onclick={ clickButton } data-
method="createUser">Create</button>
<button class="form__btn form__btn--delete" onclick={ clickButton } data-
method="deleteUser">Delete</button>
<button class="form__btn form__btn--update" onclick={ clickButton } data-
method="updateUser">Update</button>
```

Кнопки, как и поля, содержат пользовательские атрибуты, которые в кнопках называются **data-method** и содержат название вызываемого метода для соответствующей кнопки. Кроме этого, они также содержат атрибут **onclick**, который ссылается на функцию **clickButton** при срабатывании события **click** для кнопки.

Рассмотрим эту функцию подробнее:

```
// обработчик события нажатия на кнопку
clickButton(e) {
  e.preventDefault()
  e.preventUpdate = true
  this.user[e.target.dataset.method]()
}
```

В самом начале этой функции, мы останавливаем действие по умолчанию для кнопки, вызывая метод **preventDefault** объекта события **event**, который для краткости у нас называется **e**:

```
e.preventDefault()
```

Вторая команда функции **clickButton** служит для той же цели, что и такая же команда в функции **inputValue**. Она отменяет автоматический вызов обработчика обновления компонента:

```
e.preventUpdate = true
```

Последней командой функции, является:

```
this.user[e.target.dataset.method]()
```

Давайте пошагово изучим, что она делает.

В квадратных скобках:

```
[e.target.dataset.method]
```

происходит обращение к кнопке на которой сработало событие и осуществляется доступ к её пользовательскому атрибуту **data-method** для получения его значения, которое, является названием вызываемой функции из модели данных **User**. На эту модель данных мы ссылаемся через свойство:

```
this.user
```

которое доступно нам благодаря тому, как упоминалось ранее, что мы вынесли модель данных **User** в общую примесь **user** в файле **App.js**.

Каждая кнопка выполняет определённое действие. Первая кнопка создаёт нового пользователя, вторая его удаляет, а третья обновляет о нём данные, вводимые в текстовые поля.

При нажатии на кнопку, вызывается её обработчик события **clickButton**, в котором и происходит вызов соответствующей функции из модели данных **User**.

Давайте создадим эти функции в нашей модели данных.

Откройте файл **User.js** из папки **models** и добавьте в объект модели данных **User** три новых метода, сразу после метода получения списка пользователей **getUsers**:

```
// создание нового пользователя
createUser() {
  fetch('https://rem-rest-api.herokuapp.com/api/users/', {
    method: 'POST',
    credentials: 'include',
    body: JSON.stringify(this.current)
  })
  .then(response => this.trigger('home'))
}

// удаление текущего пользователя
deleteUser() {
  fetch('https://rem-rest-api.herokuapp.com/api/users/' + this.current.id, {
    method: 'DELETE',
    credentials: 'include'
  })
```

```
  .then(response => this.trigger('home'))
}

// обновление текущего пользователя
updateUser() {
  fetch('https://rem-rest-api.herokuapp.com/api/users/' + this.current.id, {
    method: 'PUT',
    credentials: 'include',
    body: JSON.stringify(this.current)
  })
  .then(response => this.trigger('home'))
}
```

Структура данных функций должна быть вам уже знакома, на примере функции получения списка пользователей **getUsers**, которую мы рассматривали в четвёртом уроке, когда создавали модуль хранения состояния. Стоит обратить внимание, что при вызове метода **fetch** в каждой функции, в объекте его параметров, который является вторым аргументом, мы указываем соответствующий метод **HTTP**:

- **POST** - создание
- **DELETE** - удаление
- **PUT** - обновление

Кроме этого, в методах создания (**createUser**) и обновления (**updateUser**), в объекте параметров **fetch**, мы также передаём данные о пользователе в свойстве **body**:

```
body: JSON.stringify(this.current)
```

Предварительно преобразуя объект **current** нашей модели данных, который содержит данные о текущем пользователе, в строку **JSON**.

Конечной точкой для метода **createUser**, как и для метода **getUsers**, является **users**:

```
'https://rem-rest-api.herokuapp.com/api/users/'
```

А вот для методов **deleteUser** и **updateUser**, к этой конечной точке добавляется **id** текущего пользователя из объекта **current** нашей модели данных:

```
'https://rem-rest-api.herokuapp.com/api/users/' + this.current.id
```

В отличие от метода **getUsers**, все три новых метода не присваивают никаких данных свойствам модели. После успешного завершения своих запросов, они вызывают событие **home**:

```
.then(response => this.trigger('home'))
```

вместо события **updated**, которое вызывается в методе **getUsers**.

Давайте рассмотрим обработчик события **home**, в нашем компоненте **UserForm**:

```
// переходим на страницу list (список пользователей)
// при получении события home от модели данных
this.user.one('home', () => this.route('list'))
```

Из комментария понятно основное его предназначение. После того, как методы модели данных успешно выполнили возложенные на них действия, они вызывают этот обработчик, который, в свою очередь, просто перенаправляет наше приложение на страницу списка пользователей.

Мы ещё не создавали маршрутизацию, мы это сделаем в следующем и последнем уроке этого руководства.

Но обратите внимание на:

```
this.route('list'))
```

Свойство **route** компонента **UserForm** является ссылкой на маршрутизатор, который мы передадим в общую примесь **route** в файле **App.js**. Точно так же, как перед этим мы передавали конструктор модели данных **User**.

> Как мы помним, наша модель данных **User** является наблюдаемой и доступна благодаря примесям во всех компонентах нашего приложения.

Давайте сделаем это сейчас. И начнём мы с подключения маршрутизатора, который в **Riot.js** называется **riot-route**.

Откройте файл **App.js** и в самом верху, после подключения **Riot.js**, добавьте подключение маршрутизатора:

```
// подключаем Riot.js
import riot from 'riot'

// подключаем Маршрутизатор
import route from '../node_modules/riot-route/dist/amd.route+tag.min'
```

> Мы будем использовать маршрутизацию на основе тегов. Данный тип маршрутизатора располагается в папке **dist**, каталога основного маршрутизатора **riot-route**.

Теперь добавим этот маршрутизатор в общую примесь **route**, там, где мы создавали примесь **user**, которой передавали конструктор модели данных **User**:

```
// создаём общую примесь user и передаём в конструктор модели данных User
// ссылку на библиотеку Riot.js, в виде аргумента riot
// вторым свойством в объекте примесей создаём ещё одну примесь,
```

```
// которая называется route и ссылается на подключенный выше маршрутизатор
riot.mixin({ user: new User(riot), route: route })
```

Вернёмся к нашему компоненту **UserForm** и рассмотрим следующий обработчик:

```
// обработчик события маршрутизатора
this.on('route', (id) => this.user.getUser(id))
```

Всем компонентам, которые используются в маршрутизации, доступно событие **route**, которое выполняется при переходе к этому компоненту по ссылке, либо при нажатии кнопок **назад** и **вперёд** в браузере. У нас таких компонентов всего два, это **UserList** и **UserForm**, который мы сейчас и изучаем.

Мы пока не создавали маршрутизацию в нашем приложении, но пример того, как она будет выглядеть, мы рассмотрим сейчас:

```
<main data-is="router">
  <route path="list"><r-list /></route>
  <route path="edit/*"><r-form /></route>
</main>
```

Мы используем маршрутизацию на основе тегов, и в качестве самого маршрутизатора у нас выступает html-элемент **main**. Он имеет два маршрута:

```
<route path="list"><r-list /></route>
<route path="edit/*"><r-form /></route>
```

Маршрут `list` ссылается на компонент **UserList**, который представляет список пользователей. А маршрут `edit/*` будет ссылаться и загружать компонент **UserForm**, который будет содержать данные того пользователя, на котором мы кликнули в списке пользователей. Звёздочка в этом маршруте является подстановочным символом и соответствует регулярному выражению:

```
([^/?#]+?)
```

Теперь ещё раз рассмотрим обработчик маршрутизатора:

```
this.on('route', (id) => this.user.getUser(id))
```

Как только мы кликнули по какому-то пользователю из списка компонента **UserList**, то сразу загружается компонент **UserForm** с данными этого пользователя и выполняется обработчик маршрутизатора. Параметр **id** в этом обработчике ссылается на подстановочный символ звёздочки в маршруте `edit/*`, и представляет собой **id** пользователя из компонента **UserList**:

```
 <!-- добавляем ссылки на пользователей -->
<a href="#!/edit/{ id }"
```

И если мы посмотрим в адресную строку браузера после щелчка мышкой на каком-либо пользователе из списка, ты мы можем увидеть:

```
#!/edit/1
#!/edit/2
или
#!/edit/37
```

Вид зависит от **id** текущего пользователя, по которому мы кликнули.

Обработчик маршрутизатора вызывает метод **getUser** нашей модели данных **User** и передаёт ему **id** текущего пользователя:

```
this.user.getUser(id))
```

Метод **getUser** возвращает данные конкретного пользователя, которые отображаются в нашем компоненте **UserForm** и, которые, мы можем изменять, удалять или вводить в поля ввода новые данные и создавать на основе этих данных новых пользователей.

Мы ещё не создавали этот метод, поэтому добавим его сейчас.

Откройте файл **User.js** и добавьте метод **getUser** в нашу модель данных:

```
// получение конкретного пользователя
getUser(id) {
  fetch('https://rem-rest-api.herokuapp.com/api/users/' + id, {
    method: 'GET',
    credentials: 'include'
  })
    .then(response => response.json())
    .then(result => {
      this.current = result
      this.trigger('updated')
    })
}
```

В этом методе мы получаем данные того пользователя, **id** которого мы ему передали в обработчике маршрутизатора. После получения данных от сервера, как и метод **getUsers**, данный метод преобразует полученный **JSON** в объект **JavaScript** и присваивает его объекту **current** нашей модели данных. После чего, он вызывает событие **updated** в нашем компоненте **UserForm**.

Давайте рассмотрим обработчик этого события:

```
 // запускаем событие обновления компонента (this.update)
// при получении события updated от модели данных
this.user.one('updated', this.update)
```

Данный обработчик просто запускает в принудительном порядке событие обновления компонента:

```
// обработчик события обновления компонента
this.on('update', () => {
  this.firstName = this.user.current.firstName
  this.lastName = this.user.current.lastName
})
```

В событии обновления компонента, мы присваиваем свойствам **firstName** и **lastName** компонента **UserForm**, данные из объекта **current** нашей модели данных **User**. Как мы помним, в объект **current** их поместил метод **getUser**, который мы перед этим рассматривали.

Свойства **firstName** и **lastName** выводят свои значения в поля ввода, в шаблоне нашего компонента **UserForm**:

свойство **{ firstName }**

```
<input type="text" class="form__input" oninput={ inputValue } data-name="firstName"
placeholder="First name" value="{ firstName }">
```

и свойство **{ lastName }**

```
<input type="text" class="form__input" oninput={ inputValue } data-name="lastName"
placeholder="Last name" value="{ lastName }">
```

Мы рассмотрели самый сложный компонент нашего приложения. В последнем уроке мы добавим маршрутизацию и подведём краткий итог проделанной нами работы.

Добавляем маршрутизацию

Это заключительный урок по созданию простого приложения в **Riot.js**. В нём мы добавим маршрутизацию к нашему приложению и подведём краткий итог проделанной нами работы.

На прошлом уроке, мы уже немного коснулись темы маршрутизации и даже подключили маршрутизатор в файле **App.js**:

```
// подключаем Маршрутизатор
import route from '../node_modules/riot-route/dist/amd.route+tag.min'
```

Напомню лишь, что мы будем использовать маршрутизацию на основе тегов.

Откройте файл главного компонента нашего приложения **App.tag**.

Вот так он выглядит у нас сейчас:

```
<app>

  <!--  подключаем компонент Header  -->
  <header data-is="r-header" />

  <!--  подключаем компонент UserList  -->
  <r-list />

  <!--  подключаем компонент Footer  -->
  <footer data-is="r-footer" />

  <!-- добавляем стили -->
  <style type="plain">
    :scope {
      display: flex;
      flex-direction: column;
      font: normal 16px Verdana;
    }
  </style>

</app>
```

Мы оставим в нём всё как есть, кроме добавления стилей для элемента маршрутизации и секции подключения компонента **UserList**:

```
<!--  подключаем компонент UserList  -->
<r-list />
```

Удалите эту секцию и на её месте добавьте:

```
<!--  подключаем маршрутизатор в тег main  -->
<main data-is="router">
  <route path="list"><r-list /></route>
  <route path="edit/*"><r-form /></route>
</main>
```

Теперь добавим немного стилей для элемента маршрутизации **main**:

```
<!-- добавляем стили -->
<style type="plain">
  :scope {
    display: flex;
    flex-direction: column;
    font: normal 16px Verdana;
  }
```

```
  /* стили для элемента маршрутизатора */
  main {
    margin-bottom: 20px;
    padding: 0 15px;
  }
</style>
```

Итоговый вид нашего файла **App.tag**:

```
<app>

  <!--  подключаем компонент Header  -->
  <header data-is="r-header" />

  <!--  подключаем маршрутизатор в тег main  -->
  <main data-is="router">
    <route path="list"><r-list /></route>
    <route path="edit/*"><r-form /></route>
  </main>

  <!--  подключаем компонент Footer  -->
  <footer data-is="r-footer" />

  <!-- добавляем стили -->
  <style type="plain">
    :scope {
      display: flex;
      flex-direction: column;
      font: normal 16px Verdana;
    }
    /* стили для элемента маршрутизатора */
    main {
      margin-bottom: 20px;
      padding: 0 15px;
    }
  </style>

</app>
```

Разберём, что мы изменили подробнее. Во-первых, мы удалили компонент **UserList** и вместо него, мы создали html-элемент **main** и добавили в него маршрутизатор с помощью атрибута **data-is** со значением **router**:

```
<main data-is="router">
```

Html-элемент **main** используется для основного содержимого страницы в **HTML5**. Во-вторых, мы сделали его маршрутизатором нашего приложения. Мы могли бы добавить маршрутизатор и так:

```
<router>
  <route path="list"><r-list /></route>
```

```
  <route path="edit/*"><r-form /></route>
</router>
```

Но мы хотели бы следовать семантике и поэтому, было решено использовать стандартный html-элемент **main** из **HTML5**.

Наш маршрутизатор содержит два тега маршрута **route**:

```
<route path="list"><r-list /></route>
<route path="edit/*"><r-form /></route>
```

Каждый из этих тегов имеет атрибут **path**, в котором содержится путь данного маршрута. Кроме этого, между открывающим и закрывающим тегами **route**, содержится подключение определённого компонента.

Для маршрута `list` будет подключаться компонент **UserList**:

```
<route path="list"><r-list /></route>
```

а для маршрута `edit/*`, компонент **UserForm**:

```
<route path="edit/*"><r-form /></route>
```

Напомню, что звёздочка в маршруте является подстановочным символом и соответствует регулярному выражению:

```
([^/?#]+?)
```

т.е. может соответствовать только букве, цифре или нижнему подчёркиванию.

Если мы сейчас запустим наше приложение:

```
npm run dev
```

то не увидим никакого списка на странице. Это связано с тем, что мы задали для нашего списка маршрут `list`.

Давайте сделаем его маршрутом по умолчанию, чтобы при открытии приложения, его автоматически перекидывало на этот маршрут. Ко всему прочему, мы добавим **#!** (hashbang) в качестве базовой части **url**. По умолчанию, базовым значением является **#**. Можно было бы оставить и так, но в **SPA** принято использовать **hashbang**.

Откройте файл **App.js** и добавьте в его конце:

```
// задаём hashbang в качестве базовой части url
route.base('#!/')

// задаём маршрут list в качестве маршрута по умолчанию
// с которого начинается открытие приложения
route('list')
```

Итоговый вид файла **App.js**:

```
// подключаем Riot.js
import riot from 'riot'

// подключаем Маршрутизатор
import route from '../node_modules/riot-route/dist/amd.route+tag.min'

// подключаем модель данных User
import User from './models/User'

// подключаем компонент UserList
import './views/UserList.tag'

// подключаем компонент App
import './views/App.tag'

// подключаем компонент Menu
import './views/Menu.tag'

// подключаем компонент Header
import './views/Header.tag'

// подключаем компонент Footer
import './views/Footer.tag'

// подключаем компонент UserForm
import './views/UserForm.tag'

// подключаем внешние стили
import './sass/styles.scss'

// создаём общую примесь user и передаём в конструктор модели данных User
// ссылку на библиотеку Riot.js, в виде аргумента riot
// вторым свойством в объекте примесей создаём ещё одну примесь,
// которая называется route и ссылается на подключенный выше маршрутизатор
riot.mixin({ user: new User(riot), route: route })

// монтируем компонент App
riot.mount('app')

// задаём hashbang в качестве базовой части url
route.base('#!/')

// задаём маршрут list в качестве маршрута по умолчанию
// с которого начинается открытие приложения
route('list')
```

Теперь, наше приложение при открытии будет автоматически перенаправлено по маршруту `list`, по которому, как мы помним, будет загружаться компонент **UserList**, который представляет список из 10 пользователей. Давайте увеличим их количесто до 100.

Откройте файл **User.js**, который представляет модуль нашей модели данных, и в его методе **getUsers** добавьте параметр **limit** со значением **100** в адрес запроса метода **fetch**:

```
fetch('https://rem-rest-api.herokuapp.com/api/users?limit=100'
```

т.е. мы добавили к запросу строку:

```
?limit=100
```

Метод **getUsers** должен теперь выглядеть так:

```
// получаем список пользователей с сервера
getUsers() {
  fetch('https://rem-rest-api.herokuapp.com/api/users?limit=100', {
    method: 'GET',
    credentials: 'include'
  })
    .then(response => response.json())
    .then(result => {
      // присваиваем результат ответа сервера свойству list модели данных
      this.list = result.data
      // запускаем событие updated, после успешного получения данных от сервера
      this.trigger('updated')
    })
}
```

> На самом деле, пользователей в базе всего 25. Может быть со временем это количество будет увеличено, поэтому мы взяли с запасом.

Мы можем снова запустить наше приложение и оно откроется по маршруту `list`, со списком из 25 пользователей. Можно щелкнуть по любому пользователю и тогда, мы перейдём на страницу редактирования этого пользователя, которую представляет компонент **UserForm** по маршруту `edit/*`.

Поредактируйте данные пользователей и поиграйте с кнопками на этой странице. После нажатия на любую кнопку, приложение будет перенаправлено по маршруту `list`, обратно к списку пользователей, как мы помним из прошлого урока:

```
// переходим на страницу list (список пользователей)
// при получении события home от модели данных
this.user.one('home', () => this.route('list'))
```

Где вы сможете увидеть сделанные вами изменения.

> Все сделанные вами изменения, никак не влияют на базу данных расположенную на сервере. Они сохраняются в куках на вашем копьютере и теряются после закрытия браузера.

Нам осталось запустить **Webpack** в режиме продакшена:

```
npm run build
```

После этого, мы можем передать файл **index.html** и папку **dist**, нашему заказчику приложения.

На этом всё!

*Мы проделали большой путь, чтобы создать это простое приложение. Но зато, мы прошли весь основной процесс создания приложений в **Riot.js**. Захватили многие моменты связанные с **Webpack** и его модулями. Надеюсь, что вам понравилось данное руководство и что всё у вас получилось! **Удачи!***