Что такое Components?

Components (или сокращённо - CompoJS) — это реактивная JavaScript-библиотека на основе стандартных Веб-компонентов, предназначенная для быстрого создания веб-приложений клиентского уровня. Библиотека распространяется под лицензией MIT, имеет небольшой размер и не требует много времени для её изучения. Чтобы начать работать с ней, достаточно иметь базовые знания HTML, CSS и JavaScript.

Ниже показан пример простого компонента с заголовком и тремя доступными в библиотеке циклами:

```
<my-component>
  <h1>${ message }</h1>

  <div c-for="obj of users">
    <div class="user">
      <p>
        <b>Имя</b>: ${ obj.name }
      </p>
      <p>
        <b>Возраст</b>: ${ obj.age }
      </p>
      <div c-for="category in obj.skills">
        <b>${ category[0].toUpperCase() + category.slice(1) }</b>:
        <ol c-for="i = 0; i < obj.skills[category].length; i++">
          <li>${ obj.skills[category][i] }</li>
        </ol>
      </div>
```

```
    </div>
    <hr>
  </div>

  <style>
    h1 {
      margin-bottom: 40px;
      color: ${ color() };
    }
    .user {
      margin: 30px 0;
    }
    hr:last-of-type {
      display: none;
    }
  </style>

  <script>
    this.message = 'Пользователи'
    this.color = () => 'orangered'
    this.users = [
      {
        name: 'Дмитрий',
        age: 28,
        skills: {
          frontend: ['HTML', 'CSS'],
          backend: ['PHP', 'Ruby', 'MySQL']
        }
      },
      {
        name: 'Ольга',
        age: 25,
        skills: {
          frontend: ['HTML', 'JavaScript'],
          backend: ['PHP']
```

```
        }
      },
      {
        name: 'Максим',
        age: 30,
        skills: {
          frontend: ['HTML', 'CSS', 'JavaScript', 'jQuery'],
          backend: ['Ruby', 'MySQL']
        }
      }
    ]
  </script>
</my-component>
```

Библиотека CompoJS создана с применением современных достижений в области JavaScript и Веб-технологий. Поэтому необходимо убедиться, чтобы ваш браузер поддерживал все нижеперечисленные стандарты:

- Web Components
- MutationObserver
- CustomEvent
- Proxy и Reflect
- Шаблонные строки
- Стрелочные функции
- Деструктуризация
- Генераторы

- pushState()
- Fetch

Кроме удобного способа создания Веб-компонентов и добавления им реактивности с помощью прокси, библиотека использует в качестве наблюдателя систему событий доступную в браузере, и предоставляет маршрутизатор, на основе наблюдателя и метода pushState(). Насколько ваш браузер соответствует заявленным выше стандартам, вы можете узнать на сайте caniuse.com.

Попробовать CompoJS в действии, можно уже прямо сейчас. Мы создадим простое приложение с компонентом показанным выше, но воспользуемся его встроенным вариантом. Компоненты в CompoJS бывают двух видов: внешние и встроенные.

Создайте папку с любым именем, скачайте в неё файл compo.min.js и добавьте в эту папку файл **index.html**:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>CompoJS</title>
</head>
<body>
  <my-component></my-component>

  <template title="my-component">
    <h1>${ message }</h1>

    <div c-for="obj of users">
      <div class="user">
        <p>
```

```
      <b>Имя</b>: ${ obj.name }
    </p>
    <p>
      <b>Возраст</b>: ${ obj.age }
    </p>
    <div c-for="category in obj.skills">
      <b>${ category[0].toUpperCase() + category.slice(1) }</b>:
      <ol c-for="i = 0; i < obj.skills[category].length; i++">
        <li>${ obj.skills[category][i] }</li>
      </ol>
    </div>
  </div>
  <hr>
</div>

<style>
  h1 {
    margin-bottom: 40px;
    color: ${ color() };
  }
  .user {
    margin: 30px 0;
  }
  hr:last-of-type {
    display: none;
  }
</style>

<script>
  this.message = 'Пользователи'
  this.color = () => 'orangered'
  this.users = [
    {
      name: 'Дмитрий',
      age: 28,
```

```
          skills: {
            frontend: ['HTML', 'CSS'],
            backend: ['PHP', 'Ruby', 'MySQL']
          }
        },
        {
          name: 'Ольга',
          age: 25,
          skills: {
            frontend: ['HTML', 'JavaScript'],
            backend: ['PHP']
          }
        },
        {
          name: 'Максим',
          age: 30,
          skills: {
            frontend: ['HTML', 'CSS', 'JavaScript', 'jQuery'],
            backend: ['Ruby', 'MySQL']
          }
        }
      ]
    </script>
  </template>

  <script src="compo.min.js"></script>
  <script>
    Compo(document.querySelector('template[title="my-component"]').outerHTML)
  </script>
</body>
</html>
```

Сохраните файл и откройте его в своём браузере. Если ваш браузер соответствует требованиям CompoJS, то на экране вы увидите:

Browser

Пользователи

Имя: Дмитрий

Возраст: 28

Frontend:

1. HTML
2. CSS

Backend:

1. PHP
2. Ruby
3. MySQL

Имя: Ольга

Возраст: 25

Frontend:

1. HTML
2. JavaScript

Backend:

1. PHP

Имя: Максим

Возраст: 30

Frontend:

1. HTML
2. CSS
3. JavaScript
4. jQuery

Backend:

1. Ruby
2. MySQL

Все примеры на этом сайте были выполнены в виде компонентов CompoJS . Если вы не видите их на экране, то ваш браузер пока не готов к работе с этой библиотекой. Это напрямую касается, например, браузера Internet Explorer.

Заключение

Библиотека CompoJS была создана без оглядки на прошлое, что объясняет отсутствие поддержки её старыми браузерами. Целью её появления было сделать разработку приложений простой и доступной каждому, что стало бы невозможно без использования современных Веб-технологий. Если вы хотите узнать о CompoJS больше, то начните её изучение с учебника, в котором на простых примерах представлена вся её доступная функциональность.

Начало работы

Перед тем, как начать работать с библиотекой CompoJS, её небходимо подключить к главному файлу вашего приложения. Как правило, таким файлом выступает файл **index.html**. Мы начнём с создания папки для проекта, в котором и будет располагаться указанный файл.

Создайте каталог **app** и скачайте в него файл библиотеки compo.min.js. Затем добавьте в этот каталог, главный файл **index.html** нашего приложения:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования компонента 'my-component' -->
  <my-component></my-component>

  <!-- тег шаблона компонента 'my-component' -->
  <template title="my-component">
    <h1>${ message }</h1>

    <style>
      h1 {
```

```
        color: ${ color() };
      }
    </style>

    <script>
      this.message = 'Мой компонент!'
      this.color = () => 'orangered'
    </script>
  </template>

  <!-- подключить CompoJS -->
  <script src="compo.min.js"></script>

  <!-- передать CompoJS шаблон компонента -->
  <script>
    Compo(document.querySelector('template[title="my-component"]').outerHTML)
  </script>
</body>
</html>
```

Откройте файл в браузере и если он поддерживает заявленные требования CompoJS, то вы должны будете увидеть на экране:

Browser

Мой компонент!

Готово! Мы только что создали первое приложение с помощью CompoJS. Давайте разберём подробнее, что происходит в файле **index.html** нашего приложения. После открывающего тега <body>, располагается тег монтирования компонента <my-component>. Этих тегов может быть любое количество на странице. Один и тот же компонент можно многократно монтировать как в главном файле приложения, так и в любом его компоненте.

За ним следует шаблон компонента, который определяется тегом <template> с атрибутом title , значением которого является имя создаваемого компонента my-component . После шаблона идёт тег <script>, который подключает CompoJS к приложению. В конце главного файла находится ещё один тег <script>, в котором глобальной функции Compo() библиотеки CompoJS, передаётся полное HTML-содержимое шаблона нашего компонента с помощью свойства outerHTML, включая и сам тег <template>.

Содержимое шаблона компонента будет подробно рассмотрено немного позже, сейчас важно понять общую схему создания приложений в CompoJS. Мы создали встроенный компонент с именем my-component . Компоненты в CompoJS можно создавать во внешних файлах, и вскоре мы увидим, как создаются внешние компоненты, а пока давайте продолжим работать со встроенными.

Если мы создадим большое количество встроенных компонентов, то выбирать каждый из них и передавать их содержимое глобальной функции Compo() , будет не очень удачным решением. Лучше выбрать все шаблоны компонентов за один раз и передать этой функции их содержимое.

Внесите изменения в файл **index.html**, как показано ниже:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
```

```
</head>
<body>
  <!-- тег монтирования компонента 'my-component1' -->
  <my-component1></my-component1>

  <!-- тег монтирования компонента 'my-component2' -->
  <my-component2></my-component2>

  <!-- тег шаблона компонента 'my-component1' -->
  <template title="my-component1">
    <h1>${ message }</h1>

    <script>
      this.message = 'Компонент 1'
    </script>
  </template>

  <!-- тег шаблона компонента 'my-component2' -->
  <template title="my-component2">
    <h1>${ message }</h1>

    <script>
      this.message = 'Компонент 2'
    </script>
  </template>

  <!-- подключить CompoJS -->
  <script src="compo.min.js"></script>

  <!-- передать CompoJS шаблоны всех компонентов -->
  <script>
    Compo([...document.querySelectorAll('template[title]')].map(temp => temp.outerHTML).join(''))
  </script>
</body>
</html>
```

Глобальная функция Compo() получает простое текстовое содержимое и создаёт из него компоненты. В этом примере мы выбрали все теги <template> с атрибутом title , с помощью метода querySelectorAll() объекта document . После этого, мы пропустили их через метод map(), который вернул массив с их полным HTML-содержимым, включая и сами теги шаблонов <template>. В конце команды мы преобразуем возвращённый массив в обычный текст с помощью метода join() и, передаём его функции Compo() для создания компонентов.

В браузере будет показан следующий результат:

Browser

Компонент 1

Компонент 2

Перед тем, как перейти к рассмотрению внешних компонентов, давай разберём ещё один способ их создания, а точнее — подмену компонентов. Внесите изменения в файл **index.html**, как показано ниже:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
```

```
<body>
  <!-- монтируем компонент 'my-component' в тег 'header' используя атрибут 'is' -->
  <header is="my-component"></header>

  <!-- тег шаблона компонента 'my-component' с атрибутом 'to' и значением 'header' -->
  <template title="my-component" to="header">
    <h1>${ message }</h1>

    <script>
      this.message = 'Заголовок'
    </script>
  </template>

  <!-- подключить CompoJS -->
  <script src="compo.min.js"></script>

  <!-- передать CompoJS шаблон компонента -->
  <script>
    Compo(document.querySelector('template[title="my-component"]').outerHTML)
  </script>
</body>
</html>
```

Используя атрибут `to` шаблона компонента, мы указываем в нём имя тега, в который мы хотим смонтировать наш компонент <my-component>. В самом же теге <header> используется его атрибут `is` со значением, соответствующем имени монтируемого компонента.

Таким образом, мы можем монтировать наши компоненты в большинство стандартных HTML-элементов:

Browser

Заголовок

По умолчанию, все компоненты создаются с открытым теневым DOM. Чтобы создать компонент с закрытым теневым деревом, достаточно в тег шаблона компонента добавить атрибут closed без значения, например:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования закрытого компонента 'my-component' -->
  <my-component></my-component>

  <!-- тег шаблона закрытого компонента 'my-component' с атрибутом 'closed' -->
  <template title="my-component" closed>
    <h1>${ message }</h1>

    <style>
      h1 {
        color: ${ color() };
      }
    </style>
```

```
    <script>
      this.message = 'Мой компонент!'
      this.color = () => 'orangered'
    </script>
  </template>

  <!-- подключить CompoJS -->
  <script src="compo.min.js"></script>

  <!-- передать CompoJS шаблон компонента -->
  <script>
    Compo(document.querySelector('template[title="my-component"]').outerHTML)
  </script>
</body>
</html>
```

Всё что мы рассмотрели выше со встроенными компонентами, в равной степени относится и к внешним компонентам. Важным отличием является тот факт, что для компонентов размещённых во внешних файлах необходимо иметь установленный сервер, без которого можно обойтись при работе со встроенными компонентами.

Внешние компоненты

Для работы с внешними компонентами, нам потребуется установленный сервер. Поскольку содержимое внешних компонентов будет располагаться во внешних файлах, доступ к которым для браузера закрыт из-за соображений политики безопасности.

Наиболее простым и удобным для наших целей является lite-server, но вы можете использовать любой другой. Для работы этого сервера, вам необходимо иметь установленный в системе Node.js — среда разработки JavaScript.

Для глобальной установки lite-server , откройте терминал командной строки и введите в нём команду:

```
npm i -g lite-server
```

В зависимости от вашей операционной системы, возможно вам придётся запускать все команды установки в терминале от имени администратора и вводить пароль. Для этого добавьте в терминале ключевое слово sudo , перед любой командой установки:

```
sudo npm i -g lite-server
```

После установки lite-server , перейдите из терминала в каталог **app** или откройте терминал из этого каталога. Для запуска сервера введите в терминале команду:

```
lite-server
```

Когда она закончит своё выполнение, у вас должно открыться окно браузера по умолчанию, в котором будет показан наш файл **index.html**. Теперь при любом изменении его содержимого, браузер будет автоматически перезагружать страницу.

Вернёмся к внешним компонентам, но перед этим, давайте условимся все наши будущие компоненты именовать с префиксом c- , вместо префикса my- . Хотя, вы можете по-прежнему использовать префикс my- или любой другой, по своему усмотрению.

> Чтобы гарантировать отсутствие конфликтов имён между встроенными и пользовательскими HTML-элементами, имя компонента должно содержать дефис - . Например, my-component – это валидное имя, а mycomponent – нет.

Давайте создадим два компонента аналогичные тем, что мы создавали в первой части этого учебника. Но в этот раз мы разместим их во внешних файлах.

Создайте в папке app первый файл component1.htm:

```
<c-component1>
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 1'
  </script>
</c-component1>
```

Второй файл будет называться component2.htm:

```
<template title="c-component2">
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 2'
  </script>
</template>
```

Файлы компонентов представляют собой обычные HTML-файлы с расширением .htm . Как видно из примера выше, шаблон второго компонента ничем не отличается от уже знакомых нам шаблонов встроенных компонентов.

Что же касается первого компонента, то вместо тега <template> с атрибутом title , мы используем тег <c-component1>, который в точности соответствует названию компонента. Атрибут title ему уже не требуется, но рассмотренные ранее для тега <template> атрибуты to и closed , по-прежнему могут в нём использоваться.

Внесите изменения в файл index.html:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1></c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <c-component2></c-component2>

  <!-- тег подключения компонента 'c-component1' -->
  <template src="component1.htm"></template>

  <!-- тег подключения компонента 'c-component2' -->
  <template data-src="component2.htm"></template>

  <!-- подключить CompoJS -->
  <script src="compo.min.js"></script>
```

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

Главный файл приложения изменился, в нём больше нет тегов <template> определяющих шаблоны компонентов, поскольку мы их вынесли во внешние файлы. Их место заняли другие теги <template> с атрибутом `src` (можно использовать `data-src`), в которых указывается путь к файлам соответствующих компонентов.

Обратите внимание, что больше нет скрипта, который вызывает функцию `Compo()` и передаёт ей содержимое компонентов. CompoJS автоматически ищет теги <template> с атрибутами `src` (или `data-src`) и читает содержимое тех файлов, пути к которым в них указаны. Затем он создаёт из этого содержимого компоненты и монтирует их в приложение.

Если открыть файл **index.html** в браузере, то на экране будут показаны наши два компонента:

Browser

Компонент 1

Компонент 2

Таким образом можно создавать любое количество компонентов во внешних файла и подключать их в файле **index.html**. Но это не совсем правильно. CompoJS придётся делать запрос для каждого такого файла, что при большом количестве компонентов может замедлить общее время подготовки приложения к работе. Давайте это исправим!

Создайте в папке app файл components.htm и скопируйте в него содержимое наших двух компонентов:

```
<c-component1>
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 1'
  </script>
</c-component1>

<template title="c-component2">
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 2'
  </script>
</template>
```

Внесите изменения в файл index.html, как показано ниже:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1></c-component1>
```

```
  <!-- тег монтирования компонента 'c-component2' -->
  <c-component2></c-component2>

  <!-- тег подключения файла компонентов -->
  <template src="components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="compo.min.js"></script>
</body>
</html>
```

Мы перенесли все компоненты в один файл, который назвали components.htm. Теперь CompoJS делает всего один запрос к этому файлу, читает его содержимое и создаёт из него компоненты. Это содержимое кешируется, что ускоряет работу при повторном обращении к сайту. Единственное неудобство состоит в том, что все компоненты приходится размещать в одном файле, а для разработчика было бы удобнее располагать компоненты в разных файлах и затем собирать их в один. Чтобы не делать это вручную, далее мы автоматизируем этот процесс.

Сборка проекта

В этой части мы автоматизируем процесс сборки компонентов в один файл. Мы начнём с реструктуризации главного каталога. Удалите из каталога app файл components.htm и создайте в нём подкаталог src. В подкаталоге src создайте ещё два новых подкаталога: assets и components. Перенесите в подкаталог components файлы: component1.htm и component2.htm из каталога app, а файл compo.min.js из этого каталога, перенесите в подкаталог assets. После всех проделанных манипуляций, структура каталога app должна выглядеть так:

```
index.html
/src
  /assets
    compo.min.js
  /components
    component1.htm
    component2.htm
```

Для сборки проекта мы воспользуемся менеджером задач Gulp. Он требует установленный в системе Node.js — среда разработки JavaScript. Сначала мы должны установить этот менеджер глобально, чтобы он был доступен для всей системы сразу.

Откройте терминал командной строки и введите в нём команду:

```
npm i -g gulp-cli
```

В зависимости от вашей операционной системы, возможно вам придётся запускать все команды установки в терминале от имени администратора и вводить пароль. Для этого добавьте в терминале ключевое слово sudo , перед любой командой установки:

```
sudo npm i -g gulp-cli
```

После установки Gulp , перейдите из терминала в каталог app или откройте терминал из этого каталога. Для создания проекта введите в терминале команду:

```
npm init -y
```

Когда она закончит своё выполнение, в каталоге **app** появится файл зависимостей **package.json**:

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

Введите в терминале ещё одну команду:

```
npm i -D gulp gulp-concat browser-sync del
```

или:

```
sudo npm i -D gulp gulp-concat browser-sync del
```

После выполнения этой команды, структура каталога **app** будет иметь следующий вид:

```
index.html
package.json
package-lock.json
/node_modules
  ...
/src
  /assets
    compo.min.js
  /components
    component1.htm
    component2.htm
```

В каталоге **app** появится подкаталог **node_modules**, в котором будут располагаться зависимости пакетов и сами пакеты, которые мы только что установили. Кроме этого, будет создан файл описания **package-lock.json**, который мы здесь рассматривать не будем. Последнее, что нам осталось сделать, это добавить файл конфигурации менеджера задач Gulp .

Создайте в каталоге **app** файл **gulpfile.js**, содержимое которого показано ниже:

```
const gulp = require('gulp')
const concat = require('gulp-concat')
const browserSync = require('browser-sync').create()
const del = require('del')

function serve(done) {
  browserSync.init({ server: './' })
  done()
}

function reload(done) {
  browserSync.reload()
  done()
```

```
}

function copy() {
  return gulp.src('src/assets/**/*')
    .pipe(gulp.dest('dist'))
}

function clean() {
  return del('dist')
}

function components() {
  return gulp.src('src/components/**/*.htm')
    .pipe(concat('components.htm'))
    .pipe(gulp.dest('dist'))
}

function watch() {
  gulp.watch('index.html', gulp.series(reload))
  gulp.watch('src/assets/**/*', gulp.series(copy, reload))
  gulp.watch('src/components/**/*.htm', gulp.series(components, reload))
}

gulp.task('default', gulp.series(clean, copy, components, serve, watch))
```

Внесите изменения в файл index.html:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1></c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <c-component2></c-component2>

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

Мы изменили путь к файлу компонентов components.htm и библиотеки compo.min.js. Все они теперь будут располагаться у нас в каталоге dist. Этот каталог Gulp будет создавать автоматически и помещать в него собранный файл компонентов из каталога components. Кроме этого, всё содержимое каталога assets, в котором обычно располагаются все необходимые для работы приложения статические ресурсы: иконки, изображения, шрифты, библиотеки и другие файлы, тоже будет скопировано в каталог dist.

Для проверки работоспособности нашего приложения, введите в терминале команду:

```
gulp
```

После этого у вас откроется окно браузера по умолчанию, в котором будут показаны наши два компонента:

Browser

Компонент 1

Компонент 2

Любые изменения в файлах компонентов из каталога **components**, вызовут пересборку файла **components.htm** и автоматическую перезагрузку браузера. При добавлении или изменении файлов в каталоге **assets**, они будут немедленно скопированы в каталог **dist**.

Эту сборку можно использовать для создания любого нового проекта и чтобы каждый раз не создавать всё это заново, выделите в каталоге **app** все дочерние файлы и подкаталоги, кроме подкаталога **node_modules** и файла **package-lock.json**, и заархивируйте их в архив под названием new.zip.

Теперь перед созданием любого нового проекта, просто разархивируйте этот архив, откройте в нём свой терминал и введите команду:

```
npm i
```

или:

```
sudo npm i
```

Данная команда восстановит все зависимости нового приложения, она создаст подкаталог node_modules и файл package-lock.json. Для запуска приложения, достаточно будет снова ввести в терминале команду:

```
gulp
```

Мы прошли весь процесс создания компонентов в CompoJS и теперь пришло время познакомиться с возможностями, которые предоставляет эта библиотека. Но сначала мы узнаем, как можно получить доступ к компонентам и их свойствам в браузере.

Выбор компонентов

В этой части мы познакомимся с тем, как выбирать и выполнять различные манипуляции с компонентами. Чтобы что-то сделать с компонентом, например, изменить его содержимое, добавить новый атрибут его HTML-элементу или задать новые значения его пользовательским свойствам (данным), этот компонент необходимо выбрать.

Запустите в терминале наш проект командой:

```
gulp
```

Когда приложение будет готово к работе, откройте в браузере консоль командой Ctrl+Shift+I или через его меню. В консоли браузера введите команду:

```
document.querySelector('c-component1')
```

Эта команда отобразит в консоли первый компонент нашего приложения. Для выбора этого компонента, мы воспользовались стандартным методом querySelector() объекта document , но можно использовать любой способ поиска HTML-элементов в документе. После того, как тег монтирования <c-component1> был выбран на странице, мы можем, например, изменить HTML-содержимое его компонента.

Введите в консоли браузера команду:

```
document.querySelector('c-component1').innerHTML = 'Новое содержимое'
```

В браузере ничего не изменилось. Это связано с тем, что всё HTML-содержимое компонентов находится в их теневом DOM. Для доступа к теневому дереву, существует специальное свойство $root .

Добавим в предыдущую команду данное свойство:

```
document.querySelector('c-component1').$root.innerHTML = 'Новое содержимое'
```

Теперь содержимое первого компонента изменилось:

Browser

Новое содержимое

Компонент 2

Свойство $root предоставляет доступ к корню теневого DOM компонента, в котором находится всё его HTML-содержимое. Мы можем сочетать с этим свойством такие методы поиска, как например: querySelector() и querySelectorAll() . Давайте изменим у второго компонента содержимое его тега <h1>, применив для его выбора метод querySelector() .

В консоли браузера введите следующую команду:

```
document.querySelector('c-component2').$root.querySelector('h1').innerText = 'Заголовок'
```

Результат работы этой команды, сразу же отразится на экране вашего браузера:

Browser

Новое содержимое

Заголовок

Данную команду можно значительно сократить, используя специальный метод компонента $() . Он заменяет собой его свойство $root и метод querySelector() , например:

```
document.querySelector('c-component2').$('h1').innerText = 'Заголовок'
```

Для замены метода querySelectorAll(), тоже существует специальный метод $$() компонента. Например, вместо:

```
document.querySelector('c-component2').$root.querySelectorAll('h1')[0].innerText = 'Заголовок'
```

можно написать команду:

```
document.querySelector('c-component2').$$('h1')[0].innerText = 'Заголовок'
```

Мы разобрались с тем, как получать доступ к компоненту в браузере и изменять его содержимое, когда это может понадобиться. Но как изменять пользовательские свойства (данные) компонента? Для доступа к пользовательским свойствам, у компонента имеется специальное свойство $data . Оно ссылается на объект данных компонента, в котором хранятся все его пользовательские свойства и методы.

У нас имеется два компонента и в каждом из них, присутствует пользовательское свойство message , которое располагается в их тегах <h1>. Но связано это свойство не с самими этими тегами, а с содержимым их текстовых узлов.

Предыдущий пример можно заменить командой:

```
document.querySelector('c-component2').$data.message = 'Заголовок'
```

Сейчас это не сработает, потому что перед этим мы удалили всё текстовое содержимое тега <h1> второго компонента и заменили его новым, с которым свойство message уже никак не связано:

```
document.querySelector('c-component2').$('h1').innerText = 'Заголовок'
```

> CompoJS является реактивной библиотекой, это означает, что любые изменения в пользовательских свойствах (данных) компонента, автоматически отражаются на связанных с ними узлах DOM. Это могут быть текстовые узлы или узлы атрибутов, в которых эти данные используются. При удалении связанных с пользовательскими свойствами узлов, реактивные связи между ними теряются.

Перезагрузите браузер и снова введите в консоли команду:

```
document.querySelector('c-component2').$data.message = 'Заголовок'
```

Теперь всё будет работать правильно:

Browser

Компонент 1

Заголовок

Всё что мы рассмотрели выше, можно использовать не только в браузере, но и в скриптах компонента. Просто вместо инструкции document.querySelector("имя_компонента") , внутри скриптов используется ключевое слово this .

Давайте посмотрим как это можно сделать, на примере первого компонента:

```
<c-component1>
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 1'

    // выбор с помощью метода querySelector() и замена текстового содержимого тега h1
    this.$root.querySelector('h1').innerText = 'Заголовок'

    // выбор с помощью метода querySelectorAll() и замена текстового содержимого тега h1
    this.$root.querySelectorAll('h1')[0].innerText = 'Заголовок'

    // выбор с помощью метода $() и замена текстового содержимого тега h1
    this.$('h1').innerText = 'Заголовок'

    // выбор с помощью метода $$() и замена текстового содержимого тега h1
    this.$$('h1')[0].innerText = 'Заголовок'

    // выбор с помощью метода $() и добавление атрибута 'title' тегу h1
    this.$('h1').setAttribute('title', 'Подсказка')

    // выбор с помощью метода $$() и удаление атрибута 'title' у тега h1
    this.$$('h1')[0].removeAttribute('title')

    // замена с помощью свойства $date текстового содержимого тега h1
    this.$data.message = 'Заголовок'

    // замена с помощью свойства $root HTML-содержимого компонента
```

```
    this.$root.innerHTML = 'Новое содержимое'
  </script>
</c-component1>
```

Таким образом, с помощью специального свойства $data , можно быстро получать доступ к любым пользовательским свойствам компонента и изменять их значения. Эти изменения будут сразу же отражаться в окне браузера, а точнее, в тех узлах DOM компонента, в которых они используются. В следующей части этого учебника, мы поближе познакомимся с объектом данных компонента.

Объект данных

Любые пользовательские свойства и методы компонента хранятся в его объекте данных, который называется $data . Все они задаются в тегах <script> его шаблона определения. Этих тегов может быть сколько угодно в шаблоне.

Внесите изменения в шаблон первого компонента:

```
<c-component1>
  <h1>${ message }</h1>

  <style>
    h1 {
      color: ${ color() };
    }
  </style>
```

```
  <script>
    // пользовательское свойство компонента
    this.message = 'Компонент 1'
  </script>

  <script>
    // пользовательский метод компонента
    this.color = () => 'orangered'
  </script>
</c-component1>
```

Заголовок первого компонента станет оранжевого цвета:

Browser

Компонент 1

Компонент 2

Со свойством message мы уже познакомились в прошлой части, когда учились получать доступ к компонентам и их свойствам в браузере. Что же касается метода color() , который размещён у нас во втором теге <script> и определяет цвет заголовка компонента, то у вас тоже не должно возникнуть с этим каких-либо трудностей, если только вы пока не знакомы со стрелочными функциями.

> Во всех представленных в учебнике примерах, JavaScript-код пишется без завершающих инструкции символа точки с запятой ; . Но вы можете их использовать при выполнении этих примеров, если вы привыкли к такому стилю написания кода, например:

```
<script>
  // пользовательское свойство компонента
  this.message = 'Компонент 1';
</script>
```

Функции в CompoJS можно использовать любые, как обычные так и стрелочные. Используя обычную функцию вместо стрелочной, код шаблона компонента из предыдущего примера будет выглядеть так:

```
<c-component1>
  <h1>${ message }</h1>

  <style>
    h1 {
      color: ${ color() };
    }
  </style>

  <script>
    // пользовательское свойство компонента
    this.message = 'Компонент 1'
  </script>

  <script>
    // пользовательский метод компонента
    this.color = function() {
      return 'orangered'
    }
```

```
  </script>
</c-component1>
```

Все пользовательские свойства и методы компонента задаются в тегах <script> его шаблона определения, с использованием ключевого слова this . На этапе создания компонента, из шаблона удаляются все его HTML-комментарии, а содержимое тегов <script> переносится в специальную функцию, которая запускается как метод объекта данных компонента.

> Свойство $data ссылается на объект данных компонента и является контекстом его выполнения, т.е. ключевым словом this упомянутой выше функции, в которой исполняется содержимое тегов <script>.

В тегах <script> можно писать любой валидный JavaScript-код. Важно лишь помнить, что без ключевого слова this , вы просто создаёте переменную, а с ним — реактивное пользовательское свойство объекта данных компонента. Например:

```
<script>
  message = 'Глобальная переменная'

  var message = 'Локальная переменная'

  let message = 'Локальная переменная'

  const message = 'Локальная константа'

  this.message = 'Пользовательское свойство'
</script>
```

Мы можем изменять любые пользовательские свойства компонентов, в том числе и свойства самих этих свойств, например: свойства пользовательских объектов или элементов массивов. Применять любые допустимые к ним методы и

эти изменения, будут сразу же отображаться в браузере.

Добавим объект и массив в шаблон первого компонента:

```
<c-component1>
  <h1>${ message }</h1>
  <pre>${ user }</pre>
  <pre>${ arr }</pre>

  <style>
    h1 {
      color: ${ color() };
    }
  </style>

  <script>
    // пользовательское свойство компонента
    this.message = 'Компонент 1'

    // пользовательский метод компонента
    this.color = () => 'orangered'

    // пользовательский объект компонента
    this.user = {
      name: 'Дмитрий',
      age: 28,
      address: {
        city: 'Москва',
        street: 'Профсоюзная'
      }
    }

    // пользовательский массив компонента
    this.arr = [1,2,3]
```

```
  </script>
</c-component1>
```

Для вывода этого объекта и массива в HTML, мы воспользовались в шаблоне компонента стандартными тегами <pre>, которые сохраняют красивое форматирование добавляемое CompoJS автоматически, при выводе любых объектов и массивов на экран.

На экране браузера мы увидим:

Browser

Компонент 1

```
{
 "name": "Дмитрий",
 "age": 28,
 "address": {
  "city": "Москва",
  "street": "Профсоюзная"
 }
}

[
 1,
 2,
 3
]
```

Компонент 2

Откройте консоль браузера и введите в ней следующие команды:

```
let comp1 = document.querySelector('c-component1')

comp1.$data.user.name = 'Иван'

comp1.$data.user.phone = '123-45-67'

comp1.$data.user.address.city = 'Тула'

comp1.$data.arr.push(4)

comp1.$data.arr.shift()

comp1.$data.arr.reverse()

comp1.$data.arr.unshift('первый')

comp1.$data.arr[1] = 58
```

Ниже показан результат применения этих команд:

Browser

Компонент 1

```
{
 "name": "Иван",
 "age": 28,
 "address": {
  "city": "Тула",
  "street": "Профсоюзная"
 },
 "phone": "123-45-67"
}
```

```
[
 "первый",
 58,
 3,
 2
]
```

Компонент 2

Благодаря прокси, можно изменять любые внутренние свойства других пользовательских объектов и элементы пользовательских массивов, и все эти изменения будут сразу же отображаться в браузере. Мы заканчиваем наше знакомство с пользовательскими свойствами и методами компонентов. Для вывода их значений в HTML-содержимое, применяются подстановки.

Подстановки

Для вывода значений пользовательских свойств в шаблоне компонента, применяются подстановки шаблонных строк. Они обозначаются знаком доллара и фигурными скобками ${ выражение } , например:

```
<!-- подстановка в содержимом тега <h1> -->
<h1>${ message }</h1>

<style>
  h1 {
    color: ${ color() }; /* подстановка в свойстве стилей */
  }
</style>
```

Подстановки можно использовать везде, где допускается вывод значений пользовательских свойств:

```
<!-- подстановка в атрибуте тега <h1> -->
<h1 title="${ message }">Компонент 1</h1>

<style>
  h1 {
    width: ${ width }px; /* подстановка в свойстве стилей перед единицей измерения */
  }
</style>
```

Пробелы между фигурными скобками в подстановках, используются лишь для улучшения их читабельности в коде:

```
<h1>${ message }</h1>
```

и они не являются обязательными:

```
<h1>${message}</h1>
```

В подстановках CompoJS можно использовать любые выражения, которые допустимы в подстановках шаблонных строк:

```
<!-- пользовательское свойство -->
${ message }

<!-- пользовательский метод -->
${ color() }

<!-- арифметическое выражение -->
${ 4 + 5 }

<!-- выражение конкатенации и т.д. -->
${ 'Имя: ' + user.name }
```

Перед именами пользовательских свойств в подстановках, можно добавлять ключевое слово this предваряемое точкой:

```
<!-- пользовательское свойство -->
${ this.message }
```

Когда внутренние методы ядра CompoJS создают реактивные связи между узлами DOM компонента и его пользовательскими свойствами, они преобразуют содержимое этих текстовых узлов и узлов атрибутов, в шаблонную строку JavaScript, обрамлённую с обоих сторон обратными кавычками `` ` ``. Поэтому вне подстановок, обратные кавычки необходимо экранировать символом обратный слеш `\`, иначе будет ошибка:

```
<!-- Браузер выдаст ошибку -->
<h1>`Компонент 1`</h1>

<!-- Ошибки не будет! -->
<h1>\`Компонент 1\`</h1>

<!-- Ошибки не будет! -->
<h1>${ `Компонент 1` }</h1>
```

Мы рассмотрели все основные особенности использования подстановок в CompoJS. Более подробную информацию об использовании шаблонных строк, вы можете прочитать в официальной документации. Далее мы рассмотрим стилизацию компонентов.

Стилизация

CompoJS полностью основан на Веб-компонентах. Это означает, что он подчиняется всем тем же правилам, которым подчиняются и сами Веб-компоненты. В частности, стилизация теневого DOM компонента в CompoJS, аналогична стилизации теневого DOM Веб-компонентов. Стили создаваемые внутри компонента, не распространяются на стили всего документа и других его компонентов.

В качестве элемента-хозяина теневого DOM, выступает тег монтирования компонента. Применённые к нему стили через селектор :host, перекрываются стилями из документа в котором он находится.

Внесите изменения в наш первый компонент:

```
<c-component1>
  <h1>${ message }</h1>

  <style>
    :host {
      display: none; /* скрываем тег монтирования (элемент-хозяин) компонента */
    }
  </style>

  <script>
    this.message = 'Компонент 1'
  </script>
</c-component1>
```

Первый компонент больше не отображается и на экране будет виден только второй компонент:

Browser

Компонент 2

Однако, стиль применённый к тегу монтирования первого компонента в самом документе, снова вернёт его на экран.

Измените файл index.html, как показано ниже:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    c-component1 {
      display: block; /* возвращаем первый компонент на экран */
    }
  </style>
</head>
<body>
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1></c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <c-component2></c-component2>
```

```
  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

Всё вышесказанное относится лишь к тегу монтирования (элементу-хозяину) компонента. Если мы в документе стилизуем тег <h1>, то стили распространятся только на находящиеся непосредственно в этом документе данные теги. На одноимённых тегах находящихся в компонентах, правила из документа никак не отразятся.

Внесите изменения в файл index.html:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    h1 {
      color: orange; /* изменит цвет всех тегов h1 только в документе */
    }
  </style>
</head>
<body>
  <!-- тег h1 документа -->
  <h1>Заголовок</h1>

  <!-- тег монтирования компонента 'c-component1' -->
```

```
  <c-component1></c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <c-component2></c-component2>

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

Удалим все стили из первого компонента:

```
<c-component1>
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 1'
  </script>
</c-component1>
```

Только тег <h1> находящийся непосредственно в документе, станет на экране оранжевого цвета:

Browser

Заголовок

Компонент 1

Компонент 2

Если мы добавим стили к тегу <h1> первого компонента:

```
<c-component1>
  <h1>${ message }</h1>

  <style>
    h1 {
      color: green; /* устанавливаем цвет тега h1 первого компонента */
    }
  </style>

  <script>
    this.message = 'Компонент 1'
  </script>
</c-component1>
```

то только его заголовок станет зелёного цвета:

Browser

Заголовок

Компонент 1

Компонент 2

Теперь давайте попробуем задать цвет фона элементу-хозяину второго компонента:

```
<template title="c-component2">
  <h1>${ message }</h1>

  <style>
    :host {
      background: cadetblue; /* задаём цвет фона элемента-хозяина второго компонента */
    }
  </style>

  <script>
    this.message = 'Компонент 2'
  </script>
</template>
```

Ничего из этого не выйдет и на экране мы увидим предыдущий результат:

Browser

Заголовок

Компонент 1

Компонент 2

Поскольку тег <c-component2> является нестандартным пользовательским тегом в HTML, то для стилизации таких его свойств, как, например, background , необходимо явно задавать ему тип отображения (свойство display) в селекторе :host , с любым подходящим значением:

```
<template title="c-component2">
  <h1>${ message }</h1>

  <style>
    :host {
      display: block; /* задаём тип отображения элемента-хозяина второго компонента */
      background: cadetblue; /* задаём фон элемента-хозяина второго компонента */
    }
  </style>
```

```
    <script>
      this.message = 'Компонент 2'
    </script>
  </template>
```

Теперь всё будет правильно работать и на экране мы увидим:

Browser

Заголовок

Компонент 1

Компонент 2

Кроме селектора :host , существуют ещё селекторы :host(selector) и :host-context(selector) . Обо всём об этом и более подробно о стилизации слотов, с которыми мы познакомимся в следующей части, вы можете узнать в руководстве по стилизации теневого DOM.

Слоты

Для передачи любого HTML-содержимого в компоненты, в CompoJS применяются слоты. В качестве примера, давайте создадим в первом компоненте слот по умолчанию и передадим в него абзац из главного файла нашего приложения.

Внесите изменения в шаблон первого компонента:

```
<c-component1>
  <h1>${ message }</h1>

  <!-- слот по умолчанию -->
  <slot></slot>

  <script>
    this.message = 'Компонент 1'
  </script>
</c-component1>
```

Мы добавили новый тег <slot> в HTML первого компонента. В этот тег будет вставляться то HTML-содержимое, которое мы передадим компоненту через его тег монтирования в файле index.html:

```
<!DOCTYPE html>
<html lang="ru">
<head>
```

```
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1>
    <!-- передать абзац в слот по умолчанию -->
    <p>Содержимое для слота по умолчанию.</p>
  </c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <!-- <c-component2></c-component2> -->

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

После передачи первому компоненту тега <p> в слот по умолчанию, браузер покажет его содержимое на экране:

Browser

Компонент 1

Содержимое для слота по умолчанию.

Мы закомментировали тег монтирования второго компонента в файле index.html, поскольку он нам сейчас не нужен. Кроме передачи всего HTML-содержимого в один слот по умолчанию, можно передавать разный HTML, в разные слоты компонента. Для этого применяется атрибут name в тегах <slot>, находящихся в самом шаблоне компонента. Данному атрибуту присваивается имя слота. А в родительских тегах передаваемого HTML-содержимого, используется атрибут slot , в котором указывается имя слота для получения этого содержимого.

Измените шаблон первого компонента, как показано ниже:

```
<c-component1>
  <h1>${ message }</h1>

  <!-- слот по умолчанию -->
  <slot></slot>

  <!-- именованный слот с именем slot1 -->
  <slot name="slot1"></slot>

  <!-- именованный слот с именем slot2 -->
  <slot name="slot2"></slot>

  <script>
    this.message = 'Компонент 1'
  </script>
</c-component1>
```

Теперь внесите изменения в файл index.html:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1>
    <!-- передать первый абзац в слот по умолчанию -->
    <p>Содержимое для слота по умолчанию.</p>

    <!-- передать второй абзац в именованный слот с именем slot1 -->
    <p slot="slot1">Содержимое для слота 1</p>

    <!-- передать третий абзац в именованный слот с именем slot2 -->
    <p slot="slot2">Содержимое для слота 2</p>
  </c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <!-- <c-component2></c-component2> -->

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

Каждый тег <p> теперь попадёт в свой, соответствующий для этого слот:

Browser

Компонент 1

Содержимое для слота по умолчанию.

Содержимое для слота 1

Содержимое для слота 2

Больше информации о слотах, вы можете получить в руководстве по работе со слотами в теневом DOM пользовательских элементов. Давайте теперь рассмотрим ещё один способ передачи данных в компоненты, который никак не связан со слотами.

Добавьте в тег монтирования первого компонента атрибут data-name , как показано ниже:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
```

```
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1 data-name="Компонент 1"></c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <!-- <c-component2></c-component2> -->

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

Измените шаблон первого компонента:

```
<c-component1>
  <!-- получение данных из атрибута data-name -->
  <h1>${ this.dataset.name }</h1>
</c-component1>
```

В браузере мы увидим уже знакомый нам результат:

Browser

Компонент 1

Мы использовали data-* атрибут в теге монтирования первого компонента, для передачи в него некоторого значения. А в шаблоне данного компонента, мы получили доступ к этому атрибуту в подстановке содержимого тега <h1>, с помощью свойства пользовательских атрибутов dataset, как показано ниже:

```
<h1>${ this.dataset.name }</h1>
```

Доступ к свойству dataset возможен только через ключевое слово this , иначе браузер выдаст ошибку. Кроме атрибутов data-* , мы можем получить доступ к значениям любых атрибутов, указанных в тегах монтирования компонентов. Например:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1 id="Компонент-1"></c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <!-- <c-component2></c-component2> -->

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>
```

```
  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

Для доступа в шаблоне компонента к атрибуту `id`, нужно перед его именем добавить ключевое слово `this`:

```
<c-component1>
  <!-- получение данных из атрибута id -->
  <h1>${ this.id }</h1>
</c-component1>
```

На экране мы увидим значение атрибута `id`, которое мы получили в подстановке содержимого тега <h1>:

Browser

Компонент-1

Однако, если у вас есть пользовательское свойство с таким же именем, как и у передаваемого атрибута, то значение будет взято из этого свойства, а не из атрибута. При этом, ключевое слово `this` можно уже не указывать:

```
<c-component1>
  <!-- получение данных из свойства id -->
  <h1>${ this.id }</h1>

  <!-- получение данных из свойства id -->
  <h1>${ id }</h1>

  <script>
    this.id = 'Свойство id'
  </script>
</c-component1>
```

На экране мы увидим значение пользовательского свойства id , выведенное дважды в HTML-содержимое компонента:

Browser

Свойство id

Свойство id

Это связано с тем, что CompoJS оборачивает исходный объект данных любого компонента в прокси. Если такого свойства в объекте данных нет, то делается запрос к свойствам (атрибутам) самого тега монтирования этого компонента. В примере выше, свойство с именем id было в компоненте, поэтому запрос к свойствам его тега монтирования не происходил.

Не смотря на это, мы легко можем получить значение атрибута `id`, как показано ниже:

```
<c-component1>
  <!-- получение данных из свойства id -->
  <h1>${ id }</h1>

  <!-- получение данных из атрибута id, используя коллекцию 'attributes' -->
  <h1>${ this.attributes['id'].value }</h1>

  <!-- получение данных из атрибута id, используя метод getAttribute() -->
  <h1>${ this.$root.host.getAttribute('id') }</h1>

  <script>
    this.id = 'Свойство id'
  </script>
</c-component1>
```

Коллекция attributes во втором теге <h1>, является стандартным свойством HTML-элементов. Мы получаем к ней доступ в шаблоне компонента, через ключевое слово `this`, как мы делали это ранее для свойства `dataset`. В последнем теге <h1> из листинга выше, используется уже знакомое нам свойство `$root`. Его присутствие необходимо для доступа к теневому DOM компонента, у которого имеется свойство host, ссылающееся на тег монтирования компонента и позволяющее получить доступ к его методу getAttribute().

> Ряд таких методов, как, например, используемый в примере `getAttribute()`, можно получить в шаблоне компонента, только через свойство `host` корня его теневого DOM, на который ссылается свойство `$root`. Если мы попытаемся вызвать их напрямую через ключевое слово `this`, то это приведёт к ошибке в браузере:

```
<!-- Ошибка, применение метода getAttribute() не через свойство 'host' -->
<h1>${ this.getAttribute('id') }</h1>
```

В браузере будет показано значение свойства id и одноимённого с ним атрибута:

Browser

Свойство id

Компонент-1

Компонент-1

О том, как передавать в компоненты любые объекты, кроме текстовых значений в атрибутах и HTML-содержимого в слотах, мы узнаем при знакомстве с наблюдателем, а пока давайте рассмотрим циклы в CompoJS.

Циклы

Циклы в CompoJS очень напоминают циклы `for` из JavaScript. Чтобы создать цикл, используется специальный атрибут `c-for` . В этом атрибуте нельзя применять подстановки вида `${ выражение }` , поскольку его содержимое вставляется в выражение цикла внутри CompoJS. Давайте добавим в первый компонент массив и пару циклов, которые будут выводить его на экран.

Внесите изменения в шаблон первого компонента:

```
<c-component1>
  <!-- вывод массива в цикле 'for' -->
  <div c-for="i = 0; i < arr.length; i++">
    <p>Элемент ${ arr[i] }</p>
  </div>

  <br>

  <!-- вывод массива в цикле 'for-of' -->
  <div c-for="item of arr">
    <p>Элемент ${ item }</p>
  </div>

  <script>
    this.arr = ['один', 'два', 'три']
  </script>
</c-component1>
```

На экране мы увидим вывод массива `arr` два раза:

Browser

Элемент один

Элемент два

Элемент три

Элемент один

Элемент два

Элемент три

Для первого вывода массива `arr` использовался обычный цикл `for`, а для второго был задействован цикл `for-of`. Независимо от типа используемого цикла, все они выводят определённое количество раз HTML-содержимое своего тега-контейнера с атрибутом `c-for`. Кроме подстановок, в атрибуте запрещено использовать ключевые слова: `var`, `let` или `const`.

Код ниже приведёт к ошибке:

```
<!-- Ошибка, ключевое слово var -->
<div c-for="var i = 0; i < arr.length; i++">...</div>

<!-- Ошибка, ключевое слово let -->
<div c-for="let i = 0; i < arr.length; i++">...</div>
```

```
<!-- Ошибка, ключевое слово const -->
<div c-for="const i = 0; i < arr.length; i++">...</div>
```

Помимо вывода простых массивов, можно ещё выводить объекты и любые другие значения, которые обычно выводятся в циклах JavaScript. Давайте выведем в первом компоненте объект и числа от 0 до 4.

Измените шаблон первого компонента, как показано ниже:

```
<c-component1>
  <!-- вывод чисел от 0 до 4 в цикле 'for' -->
  <div c-for="i = 0; i < 5; i++">
    <p>Число ${ i }</p>
  </div>

  <br>

  <!-- вывод объекта в цикле 'for-in' -->
  <div c-for="prop in user">
    <p><b>${ prop }</b> - ${ user[prop] }</p>
  </div>

  <script>
    this.user = {
      name: 'Дмитрий',
      age: 28
    }
  </script>
</c-component1>
```

Первый цикл выведет на экран числа от 0 до 4, а второй покажет объект user :

Browser

Число 0

Число 1

Число 2

Число 3

Число 4

name - Дмитрий

age - 28

Циклы можно создавать с любой глубиной вложенности:

```
<c-component1>
  <!-- вывод массива объектов в цикле 'for-of' -->
  <div c-for="obj of users">
    <div class="user">
      <p>
        <b>Имя</b>: ${ obj.name }
      </p>
```

```
    <p>
      <b>Возраст</b>: ${ obj.age }
    </p>
    <!-- вывод объекта в цикле 'for-in' -->
    <div c-for="category in obj.skills">
      <b>${ category[0].toUpperCase() + category.slice(1) }</b>:
      <!-- вывод массива в цикле 'for' -->
      <ol c-for="i = 0; i < obj.skills[category].length; i++">
        <li>${ obj.skills[category][i] }</li>
      </ol>
    </div>
  </div>
  <hr> <!-- разделитель объектов -->
</div>

<style>
  .user {
    margin: 30px 0;
  }
  hr:last-of-type {
    display: none;
  }
</style>

<script>
  this.users = [
    {
      name: 'Дмитрий',
      age: 28,
      skills: {
        frontend: ['HTML', 'CSS'],
        backend: ['PHP', 'Ruby', 'MySQL']
      }
    },
    {
```

```
        name: 'Ольга',
        age: 25,
        skills: {
          frontend: ['HTML', 'JavaScript'],
          backend: ['PHP']
        }
      },
      {
        name: 'Максим',
        age: 30,
        skills: {
          frontend: ['HTML', 'CSS', 'JavaScript', 'jQuery'],
          backend: ['Ruby', 'MySQL']
        }
      }
    ]
  </script>
</c-component1>
```

Мы добавили внутренний цикл `for-in` и тег <hr>, в содержимое внешнего цикла `for-of`. В цикл `for-in` тоже был добавлен внутренний цикл `for`. В циклах можно использовать любое HTML-содержимое, но все HTML-комментарии и пустые текстовые узлы, будут автоматически удаляться из содержимого циклов.

Созданные во время выполнения этих циклов переменные: `obj`, `category` и `i`, определяются в специальной области, где переменным внутренних циклов доступны переменные из внешних. Они не будут переопределять одноимённые переменные созданные в компоненте или в глобальном коде приложения.

На экране будет показан массив объектов пользователей:

Browser

Имя: Дмитрий

Возраст: 28

Frontend:

1. HTML
2. CSS

Backend:

1. PHP
2. Ruby
3. MySQL

Имя: Ольга

Возраст: 25

Frontend:

1. HTML
2. JavaScript

Backend:

1. PHP

Имя: Максим

Возраст: 30

Frontend:

1. HTML
2. CSS
3. JavaScript
4. jQuery

Backend:

1. Ruby
2. MySQL

Мы рассмотрели всё, что связано с циклами в CompoJS. В следующей части мы узнаем, как можно скрывать любые HTML-элементы на экране браузера, используя для этого ещё один специальный атрибут.

Скрытие

Специальный атрибут c-hide позволяет скрывать HTML-элементы компонента на странице, добавляя им стандартный атрибут hidden. Если значение содержащееся в атрибуте c-hide вычисляется как true , то элементу добавляется атрибут hidden , если как false , то атрибут hidden удаляется из элемента. В атрибуте c-hide нельзя использовать подстановки вида ${ выражение } , поскольку его содержимое оборачивается в подстановку автоматически.

Внесите изменения в шаблон первого компонента:

```
<c-component1>
  <h1 c-hide="hide">${ message }</h1>

  <script>
    this.message = 'Компонент 1'
    this.hide = false
  </script>
</c-component1>
```

Мы создали пользовательское свойство hide и присвоили ему значение false . Затем мы указали это свойство в атрибуте c-hide тега <h1> нашего компонента. Поскольку значением свойства было false , то мы увидим на экране этот заголовок:

Browser

Компонент 1

Откройте консоль браузера и введите в ней следующую команду:

```
document.querySelector('c-component1').$data.hide = true
```

Теперь тег <h1> больше не показывается на экране:

Browser

Если свойству hide снова присвоить значение false :

```
document.querySelector('c-component1').$data.hide = false
```

тег <h1> опять появится:

Browser

Компонент 1

Атрибут c-hide использует для определения своего значения, общие правила преобразования в логические типы. Это означает, что вместо значения true , можно использовать, например, цифру 1 , а значение false , заменить на пустую строку. В следующей части учебника, мы познакомимся с событиями.

События

Работа с событиями в CompoJS, аналогична работе с ними в обычном HTML. Но есть несколько особенностей, которые мы все здесь и рассмотрим. Давайте выведем на экран значение свойства message , при срабатывании браузерного события click .

Внесите изменения в шаблон первого компонента:

```
<c-component1>
  <h1 onclick="console.log(this.$data.message)">${ message }</h1>

  <script>
    this.message = 'Компонент 1'
  </script>
</c-component1>
```

Если у HTML-элемента имеются атрибуты событий начинающиеся с on , то данному элементу добавляется специальное свойство $data , которое ссылается на объект данных компонента. Используя эту особенность в CompoJS, мы можем получить доступ к пользовательским свойствам компонента, прямо в обработчике атрибута этого события, как показано в листинге выше.

Откройте приложение в браузере и кликните мышкой по заголовку **Компонент 1**:

Browser

Компонент 1

Перейдите в консоль и вы увидите там значение свойства message :

```
Компонент 1
```

Ключевое слово this в атрибутах событий, ссылается на сам HTML-элемент, в котором эти атрибуты находятся. Объект события в таких атрибутах, доступен через ключевое слово event. Например:

```
<c-component1>
  <h1 onclick="console.log(event)">${ message }</h1>

  <script>
    this.message = 'Компонент 1'
  </script>
</c-component1>
```

Вместо встроенных в элементы атрибутов событий, мы можем использовать внешние обработчики:

```
<c-component1>
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 1'

    // внешний обработчик события 'click'
    this.$('h1').addEventListener('click', event => {
      console.log(this.message)
    })
  </script>
</c-component1>
```

Обратите внимание, что для назначения обработчика HTML-элементу, мы воспользовались стандартным методом addEventListener(). Он вызывается у тега <h1>, который мы предварительно выбрали с помощью специального метода $().

Результат будет таким же, как и в первом примере, когда мы использовали атрибут события onclick . В качестве внешнего обработчика, мы применили стрелочную функцию. Ключевое слово this в таких функциях соответствует контексту, в которых они были определены. В нашем случае, оно соответствует объекту данных компонента. Поэтому нет необходимости использовать свойство $data для доступа к свойству message , как мы делали это в атрибуте события перед этим.

Если бы мы использовали обычную функцию:

```
<c-component1>
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 1'
```

```
    // внешний обработчик события 'click'
    this.$('h1').addEventListener('click', function(event) {
      console.log(this.message)
    })
  </script>
</c-component1>
```

то в консоли мы увидели бы:

```
undefined
```

Потому что ключевое слово `this` в обычных функциях назначаемых в качестве обработчиков, как и в атрибутах событий, ссылается на HTML-элемент, для которого регистрируется это событие. У элемента <h1> нет свойства `message`, равно как и нет свойства `$data`, которое добавляется только HTML-элементам имеющим атрибуты событий. Выйти из этого положения можно двумя способами.

Либо переопределить ключевое слово `this` в обработчике с помощью метода bind():

```
<c-component1>
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 1'

    // внешний обработчик события 'click'
    this.$('h1').addEventListener('click', function(event) {
      console.log(this.message)
    }.bind(this))
```

```
  </script>
</c-component1>
```

Но так мы теряем доступ к HTML-элементу, поскольку `this` теперь ссылается на объект данных компонента. Второй способ подразумевает присвоение внешней константе ссылки на этот объект:

```
<c-component1>
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 1'

    // сохраняем в константе ссылку на объект данных
    const self = this

    // внешний обработчик события 'click'
    this.$('h1').addEventListener('click', function(event) {
      console.log(self.message)
    })
  </script>

</c-component1>
```

Калькулятор

В качестве последнего упражнения с событиями, давайте напишем простой калькулятор площади. Его часто можно встретить на различных сайтах по продаже пластиковых окон.

Внесите изменения в шаблон первого компонента:

```
<c-component1>
  <form action="#">
    <p>Ширина: <input type="number" oninput="this.$data.width = event.target.value"></p>
    <p>Высота: <input type="number" oninput="this.$data.height = event.target.value"></p>
    <p>Площадь: ${ width * height }</p>
  </form>

  <script>
    this.width = 0
    this.height = 0
  </script>
</c-component1>
```

CompoJS является реактивной библиотекой. Это означает, что при изменении значения любого пользовательского свойства компонента, изменяется и содержимое тех узлов DOM, в которых это свойство используется. Например, при изменении значения пользовательского свойства `width` или `height`, содержимое последнего тега <p>, выводящего значение площади, будет автоматически пересчитываться.

Попробуйте ввести различные числовые значения в поля `Ширина` и `Высота`:

Browser

Ширина:

Высота:

Площадь: 0

Для этого примера были использованы атрибуты событий oninput , которые расположены в элементах <input>. Но мы могли бы вынести обработчики и во внешние функции, например:

```
<c-component1>
  <form action="#">
    <p>Ширина: <input id="width" type="number"></p>
    <p>Высота: <input id="height" type="number"></p>
    <p>Площадь: ${ width * height }</p>
  </form>

  <script>
    this.width = 0
    this.height = 0

    this.$('#width').addEventListener('input', event => this.width = event.target.value)
    this.$('#height').addEventListener('input', event => this.height = event.target.value)
  </script>
</c-component1>
```

Кроме этого, вместо нескольких обработчиков для элементов <input>, можно повесить один обработчик на корень теневого DOM, доступ к которому можно получить через свойство $root :

```
<c-component1>
  <form action="#">
```

```
    <p>Ширина: <input id="width" type="number"></p>
    <p>Высота: <input id="height" type="number"></p>
    <p>Площадь: ${ width * height }</p>
  </form>

  <script>
    this.width = 0
    this.height = 0

    this.$root.addEventListener('input', event => this[event.path[0].id] = event.path[0].value)
  </script>
</c-component1>
```

В этом примере, мы добавили обработчик события `input` для корневого элемента теневого DOM компонента. Как мы помним, доступ к нему можно получить через специальное свойство `$root`. А чтобы получить доступ к элементу инициировавшему событие, в нашем случае к элементу <input>, внутри такого обработчика используется свойство `path[0]` объекта события `event`.

Но представим ситуацию, что у нас есть несколько компонентов-калькуляторов и мы не хотели бы писать однотипные обработчики в каждом из них, а вместо этого, написать один обработчик для всех компонентов. Как это можно сделать?

Внесите изменения в шаблон первого компонента:

```
<c-component1>
  <h1>Компонент 1</h1>

  <form action="#">
    <p>Ширина: <input id="width" type="number"></p>
    <p>Высота: <input id="height" type="number"></p>
    <p>Площадь: ${ width * height }</p>
```

```
  </form>

  <script>
    this.width = 0
    this.height = 0
  </script>
</c-component1>
```

и в шаблон второго:

```
<template title="c-component2">
  <h1>Компонент 2</h1>

  <form action="#">
    <p>Ширина: <input id="width" type="number"></p>
    <p>Высота: <input id="height" type="number"></p>
    <p>Площадь: ${ width * height }</p>
  </form>

  <script>
    this.width = 0
    this.height = 0
  </script>
</template>
```

Теперь измените файл index.html, как показано ниже:

```
<!DOCTYPE html>
<html lang="ru">
```

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1></c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <c-component2></c-component2>

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>

  <script>
    // повесить обработчик события 'input' на объект document
    document.addEventListener('input', event => event.target.$data[event.path[0].id] = event.path[0].value)
  </script>
</body>
</html>
```

Мы добавили новый тег <script> в конце файла index.html и в нём, мы повесили обработчик события input на объект document . Внутри этого обработчика, мы получаем доступ к компоненту в котором произошло событие, с помощью свойства target объекта события event .

Затем, через специальное свойство $data (объект данных компонента), мы получаем доступ к пользовательскому свойству, название которого совпадает с id элемента <input> этого компонента, и присваиваем ему значение из свойства value

элемента <input>, в котором и произошло данное событие. Доступ к элементу <input> осуществляется с помощью свойства path[0] объекта события event .

На экране браузера мы увидим следующий результат:

Browser

Компонент 1

Ширина:

Высота:

Площадь: 0

Компонент 2

Ширина:

Высота:

Площадь: 0

Обработчик события `input` для объекта `document`, можно было бы не выносить в файл **index.html**, а создать, например, во втором компоненте:

```
<template title="c-component2">
  <h1>Компонент 2</h1>

  <form action="#">
    <p>Ширина: <input id="width" type="number"></p>
    <p>Высота: <input id="height" type="number"></p>
    <p>Площадь: ${ width * height }</p>
  </form>

  <script>
    this.width = 0
    this.height = 0

    // повесить обработчик события 'input' на объект document
    document.addEventListener('input', event => event.target.$data[event.path[0].id] = event.path[0].value)
  </script>
</template>
```

Отдельно стоит добавить, что всё вышесказанное относится лишь к открытым компонентам, которые создаются по умолчанию. Для закрытых компонентов (имеющих атрибут `closed`), свойство `path[0]` будет ссылаться на сам компонент, а не на его элемент <input>. Узнать больше...

Если вы плохо знакомы с браузерными событиями и не до конца разобрались в том, что происходило в примерах выше, то дополнительную информацию о событиях, вы сможете получить из руководства по работе с ними. В следующей части этого учебника, мы рассмотрим жизненный цикл компонента.

Жизненный цикл

Если вы плохо знакомы с Веб-компонентами, то обязательно прочтите руководство по введению в пользовательские элементы. У Веб-компонентов существуют три основных метода жизненного цикла:

1. connectedCallback() - вызывается при добавлении элемента в документ
2. disconnectedCallback() - вызывается при удалении элемента из документа
3. adoptedCallback() - вызывается при перемещении элемента в новый документ

Основные методы Веб-компонентов, напрямую в CompoJS не поддерживаются. Но существуют их короткие альтернативы:

1. $connected() - вызывается при добавлении компонента в документ
2. $disconnected() - вызывается при удалении компонента из документа
3. $adopted() - вызывается при перемещении компонента в новый документ

Рассмотрим работу с методом $connected() . Для этого, внесите изменения в наш первый компонент:

```
<c-component1>
  <h1>${ message }</h1>
```

```
  <script>
    this.message = 'Компонент 1'

    this.$connected(() => console.log(`${this.message} добавлен в документ!`))
  </script>
</c-component1>
```

Браузер выведет сообщение в консоли после того, как компонент будет добавлен в документ:

```
Компонент 1 добавлен в документ!
```

Мы можем передать методу $connected() сколько угодно функций обратных вызовов. И все они будут вызваны по очереди, после добавления компонента в документ:

```
<c-component1>
  <h1>${ message }</h1>

  <script>
    this.message = 'Компонент 1'

    this.$connected(
      () => console.log(`${this.message} добавлен в документ!`),
      function() {
        console.log(`${this.message} уже добавлен.`)
      }
    )
  </script>
</c-component1>
```

Этот пример выведет в консоли два сообщения:

```
Компонент 1 добавлен в документ!
Компонент 1 уже добавлен.
```

Остальные два метода работают аналогичным образом, но вызываются при срабатывании других событий компонента. Далее мы рассмотрим наблюдателя, основанного на системе пользовательских событий в браузере.

Наблюдатель

CompoJS использует в качестве наблюдателя, доступный в браузерах объект CustomEvent. Он позволяет создавать пользовательские события и обмениваться любыми данными между компонентами. Через слоты и атрибуты можно передавать лишь текстовые значения, в случае с наблюдателем, мы можем передавать в компоненты любые объекты.

Мы рассмотрим работу с наблюдателем, на примере наших двух компонентов и создадим ещё один, который будет имитировать работу сервера. В любом компоненте доступен специальный метод `$event()`, который представляет собой функцию-обёртку над конструктором `CustomEvent`. В первом аргументе он принимает название события в виде строки, а во втором объект параметров, аналогичный объекту параметров конструктора. Этот метод возвращает объект события создаваемый конструктором.

В папке **components** создайте новый файл **server.htm**:

```
<c-server>
  <script>
    // создать массив объектов 'users'
    const users = [
      {
        name: 'Дмитрий',
        age: 28
      },
      {
        name: 'Ольга',
        age: 25
      },
      {
        name: 'Максим',
        age: 30
      }
    ]

    // создать объект пользовательского события 'getusers' и передать ему массив объектов 'users'
    const getUsers = this.$event('getusers', {
      detail: users
    })

    // запустить событие 'getusers', через 2 секунды после монтирования компонента
    setTimeout(() => document.dispatchEvent(getUsers), 2000)
  </script>
</c-server>
```

Мы создали новый компонент <c-server>, который будет имитировать у нас работу сервера. В шаблоне этого компонента нет никаких HTML-элементов и стилей. Компонент не обязан их содержать и может состоять только из логики. Массив users будет передаваться в компонент <c-component1>, через 2 секунды после монтирования компонента <c-server>.

После определения массива `users`, мы создаём объект пользовательского события `getusers`. В качестве первого аргумента, специальному методу `$event()` передаётся название события, а во втором аргументе передаётся объект параметров со свойством detail, которому присваивается ссылка на массив `users`.

Вместо метода `$event()`, можно было бы воспользоваться конструктором `CustomEvent`, например:

```
<c-server>
  <script>
    // создать массив объектов 'users'
    const users = [
      {
        name: 'Дмитрий',
        age: 28
      },
      {
        name: 'Ольга',
        age: 25
      },
      {
        name: 'Максим',
        age: 30
      }
    ]

    // создать объект пользовательского события 'getusers' и передать ему массив объектов 'users'
    const getUsers = new CustomEvent('getusers', {
      detail: users
    })

    // запустить событие 'getusers', через 2 секунды после монтирования компонента
    setTimeout(() => document.dispatchEvent(getUsers), 2000)
  </script>
</c-server>
```

В конце мы вызываем системный метод setTimeout(), который через 2 секунды запустит наше событие getusers с помощью метода dispatchEvent() объекта document . В первом аргументе ему передаётся объект события, ссылку на который мы сохранили в константе getUsers . События должны запускаться на объекте document , чтобы они были доступны любому компоненту в приложении.

Внесите изменения в шаблон первого компонента:

```
<c-component1>
  <pre>${ users }</pre>

  <script>
    // создать свойство 'users'
    this.users = ''

    // создать обработчик для события 'getusers' и при наступлении этого события
    document.addEventListener('getusers', event => {
      // присвоить свойству `users` полученный массив объектов, через свойство 'detail' параметра 'event'
      this.users = event.detail
    })
  </script>
</c-component1>
```

Сначала мы создаём пользовательское свойство users , которое просто содержит пустую строку. Впоследствии ему будет присваиваться массив users из компонента <c-server>. Для отслеживания пользовательских событий в любом компоненте, мы используем стандартный метод addEventListener() объекта document .

После срабатывания события getusers в компоненте <c-server>, будет выполнен обработчик этого события из компонента <c-component1>. Внутри этого обработчика, через свойство detail параметра event , мы получаем доступ к созданному на сервере массиву users и присваиваем его одноимённому свойству первого компонента.

Внесите изменения в файл **index.html**, как показано ниже:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1></c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <!-- <c-component2></c-component2> -->

  <!-- тег монтирования компонента 'c-server' -->
  <c-server></c-server>

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

Мы просто добавили в этот файл, тег монтирования компонента <c-server>. Через 2 секунды после запуска приложения, мы увидим на экране браузера массив объектов users :

Browser

```
[
 {
  "name": "Дмитрий",
  "age": 28
 },
 {
  "name": "Ольга",
  "age": 25
 },
 {
  "name": "Максим",
  "age": 30
 }
]
```

Давайте раскомментируем второй компонент в файле index.html:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>Document</title>
</head>
<body>
  <!-- тег монтирования компонента 'c-component1' -->
  <c-component1></c-component1>

  <!-- тег монтирования компонента 'c-component2' -->
  <c-component2></c-component2>

  <!-- тег монтирования компонента 'c-server' -->
  <c-server></c-server>

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

Мы добавим новое событие `loadedusers` в компонент <c-component1>, которое будет вызываться сразу после того, как мы получим от компонента <c-server> массив `users` .

Внесите изменения в шаблон первого компонента:

```
<c-component1>
  <pre>${ users }</pre>

  <script>
    // создать свойство 'users'
    this.users = ''
```

```
    // создать объект пользовательского события 'loadedusers' и передать ему текстовое сообщение
    const loadedUsers = this.$event('loadedusers', {
      detail: 'Загрузка данных завершена!'
    })

    // создать обработчик для события 'getusers' и при наступлении этого события
    document.addEventListener('getusers', event => {
      // присвоить свойству `users` полученный массив объектов, через свойство 'detail' параметра 'event'
      this.users = event.detail

      // запустить событие 'loadedusers', после получения данных от сервера
      document.dispatchEvent(loadedUsers)
    })
  </script>
</c-component1>
```

Используя уже знакомый нам метод $event() , мы создали новое событие loadedusers и сохранили ссылку на его объект в константе loadedUsers . В обработчик этого события передаётся простое текстовое сообщение, о успешной загрузки данных. Событие loadedusers вызывается в обработчике события getusers после того, как от сервера будет получен массив объектов и присвоен свойству users первого компонента.

Теперь внесите изменения в шаблон второго компонента:

```
<template title="c-component2">
  <h3>${ message }</h3>

  <style>
    h3 {
      color: ${ color };
    }
```

```
  </style>

  <script>
    // создать свойство 'message'
    this.message = 'Данные загружаются...'

    // создать свойство 'color'
    this.color = 'red'

    // создать обработчик для события 'loadedusers' и при наступлении этого события
    document.addEventListener('loadedusers', event => {
      // присвоить свойству `message` полученное текстовое сообщение, через свойство 'detail' параметра 'event'
      this.message = event.detail

      // изменить значение свойства 'color'
      this.color = 'green'
    })
  </script>
</template>
```

Когда мы запустим наше приложение в браузере, то на экране мы увидим:

Browser

Данные загружаются...

Через 2 секунды сработает обработчик getusers в первом компоненте, и полученный от сервера массив объектов будет присвоен его свойству users . После этого запустится событие loadedusers , обработчик которого мы определили во втором компоненте. Он просто получает сообщение от первого компонента и присваивает его своему свойству message . Потом он меняет значение своего свойства color и полученное сообщение становится зелёного цвета:

Browser


```
[
 {
  "name": "Дмитрий",
  "age": 28
 },
 {
  "name": "Ольга",
  "age": 25
 },
 {
  "name": "Максим",
  "age": 30
 }
]
```


Загрузка данных завершена!

Мы рассмотрели основные способы использования пользовательских событий в CompoJS. Больше информации вы сможете получить из руководства по работе с ними в браузере. В последней части этого учебника мы познакомимся с

маршрутизатором, который основан на пользовательских событиях и методе pushState() .

Маршрутизатор

Маршрутизатор в CompoJS основан на рассмотренных ранее пользовательских событиях и методе pushState(). Для создания маршрутизатора, библиотека предоставляет специальный метод $router() :

```
this.$router(this.$('nav'), null,
  this.$event('/'),
  this.$event('component1'),
  this.$event('component2')
)
```

Первым аргументом данного метода указывается HTML-элемент, к которому прикрепляется создаваемый маршрутизатор. Ссылки вне этого элемента, маршрутизатором обрабатываться не будут.

Во втором аргументе метода $router() передаётся объект параметров, null или пустая строка. Передаваемый объект параметров аналогичен тому, что получает обработчик addEventListener() в своём последнем аргументе. Кроме свойств: once , capture и passive , в этот объект можно передать дополнительное свойство start со значением true , для проверки текущего маршрута приложения при запуске маршрутизатора.

Третьим, четвёртым и т.д. аргументами, их может быть сколько угодно, передаются рассмотренные в предыдущей части пользовательские объекты событий, обработчики которым тоже назначаются методом addEventListener() .

Для проверки работы маршрутизатора, мы создадим два дополнительных компонента: <c-menu> и <c-content>. В компоненте Меню будут находиться три ссылки. Первая ссылка будет выводить содержимое компонента Контент по умолчанию. Остальные две ссылки будут соответствовать ранее созданным компонентам: <c-component1> и <c-component2>, и выводить их содержимое в компонент Контент.

Создайте в папке **components** новый файл **menu.htm**:

```
<c-menu>
  <nav>
    <a href="/">Главная</a>
    <a href="component1">Компонент 1</a>
    <a href="component2">Компонент 2</a>
  </nav>

  <style>
    a {
      margin-right: 10px;
    }
  </style>

  <script>
    // добавить маршрутизатор элементу 'nav' и передать ему объекты событий
    this.$router(this.$('nav'), null,
      this.$event('/'),
      this.$event('component1'),
      this.$event('component2')
    )

    // сохранить ссылку на компонент Контент
    const content = document.querySelector('c-content')

    // определить обработчик для события '/'
```

```
    document.addEventListener('/', event => {
      // отобразить содержимое компонента Контент по умолчанию
      content.innerHTML = ''
    })

    // определить обработчик для события 'component1'
    document.addEventListener('component1', event => {
      // отобразить первый компонент в компоненте Контент
      content.innerHTML = '<c-component1></c-component1>'
    })

    // определить обработчик для события 'component2'
    document.addEventListener('component2', event => {
      // отобразить второй компонент в компоненте Контент
      content.innerHTML = '<c-component2></c-component2>'
    })
  </script>
</c-menu>
```

В самом начале логики компонента, мы вызываем рассмотренный ранее метод $router() :

```
// добавить маршрутизатор элементу 'nav' и передать ему объекты событий
this.$router(this.$('nav'), null,
  this.$event('/'),
  this.$event('component1'),
  this.$event('component2')
)
```

Первым аргументом ему передаётся HTML-элемент <nav>, который содержит ссылки на разные компоненты. Во втором аргументе передаётся значение null , поскольку никакие параметры мы задавать пока не собираемся. Последние три

аргумента являются объектами событий, возвращаемые методом $event() . Названия событий, которые мы передаём методу $event() , полностью соответствуют маршрутам в атрибутах href ссылок меню компонента:

```
<nav>
  <a href="/">Главная</a>
  <a href="component1">Компонент 1</a>
  <a href="component2">Компонент 2</a>
</nav>
```

После вызова метода $router() , мы сохраняем ссылку в константе content на тег монтирования компонента Контент, в который будут загружаться другие компоненты:

```
// сохранить ссылку на компонент Контент
const content = document.querySelector('c-content')
```

Последними идут три обработчика тех объектов событий, которые мы передали методу $router() . Эти обработчики определяются через метод addEventListener() объекта document :

```
// определить обработчик для события '/'
document.addEventListener('/', event => {
  // отобразить содержимое компонента Контент по умолчанию
  content.innerHTML = ''
})

// определить обработчик для события 'component1'
document.addEventListener('component1', event => {
  // отобразить первый компонент в компоненте Контент
```

```
  content.innerHTML = '<c-component1></c-component1>'
})

// определить обработчик для события 'component2'
document.addEventListener('component2', event => {
  // отобразить второй компонент в компоненте Контент
  content.innerHTML = '<c-component2></c-component2>'
})
```

В обработчике первого события `/` отображается содержимое компонента Контент по умолчанию, т.е. когда ему не передаётся никакой внешний HTML. А в обработчиках событий: `component1` и `component2`, мы присваиваем свойству innerHTML тега монтирования компонента Контент простую строку, содержащую теги монтирования соответствующих компонентов. Можно использовать и самозакрывающиеся теги в строке, например: <c-component1/> и <c-component2/>.

Теперь давайте создадим в папке **components** новый файл **content.htm**:

```
<c-content>
  <!-- Содержимое компонента Контент по умолчанию -->
  <slot><h1>Главная</h1></slot>
</c-content>
```

Компонент Контент содержит безымянный слот, в котором будет отображаться его HTML-содержимое по умолчанию, т.е. когда свойство `innerHTML` его тега монтирования, не содержит в себе никакой передаваемый в слот компонента HTML. Например, это будет происходить при запуске приложения в браузере или когда сработает обработчик события `/`:

```
// определить обработчик для события '/'
document.addEventListener('/', event => {
```

```
  // отобразить содержимое компонента Контент по умолчанию
  content.innerHTML = ''
})
```

Нам осталось внести изменения в шаблон первого компонента:

```
<c-component1>
  <h1>Компонент 1</h1>
</c-component1>
```

и второго:

```
<template title="c-component2">
  <h1>Компонент 2</h1>
</template>
```

И подправить главный файл приложения index.html:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
```

```
  <!-- тег монтирования компонента Меню -->
  <c-menu></c-menu>

  <!-- тег монтирования компонента Контент -->
  <c-content></c-content>

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

В нём мы просто подключаем два новых компонента: Меню и Контент. Компонент Сервер нам больше не нужен, а остальные два компонента будут подключаться у нас через маршрутизатор.

Если мы сейчас запустим наше приложение в браузере, то на экране мы увидим:

Browser

Главная Компонент 1 Компонент 2

Главная

Попробуйте пощелкать по разным ссылкам, чтобы оценить результат работы маршрутизатора. Каждая ссылка будет выводить в компонент Контент, соответствующий ей компонент из приложения.

О событиях

Передаваемые через метод `$event()` в маршрутизатор события:

```
this.$event('/'),
this.$event('component1'),
this.$event('component2')
```

представляют собой регулярные выражения в строке. Каждое такое выражение имеет следующий вид:

```
new RegExp('^[/]?событие$')
```

Это означает, что атрибуты `href` в ссылках могут содержать начальный слэш `/`, но это не является обязательным условием. Например, следующие две ссылки будут разнозначны для события `component1`:

```
<a href="/component1">Компонент 1</a>
<a href="component1">Компонент 1</a>
```

передаваемое в маршрутизатор через метод `$event()`:

```
this.$event('component1')
```

Маршрутизатор не обрабатывает ссылки на файлы: *.jpg*, *.png*, *.zip*, *.html* и все остальные, а также игнорирует внешние ссылки, ведущие на другие сайты. Кроме этого, само событие должно указываться либо полностью в методе $event() и его обработчике, т.е. соответствовать значению атрибута href , связанной с этим событием ссылки. Либо событие может содержать в себе специальные символы, как показано в примере ниже:

```
// добавить маршрутизатор элементу 'nav'
this.$router(this.$('nav'), null,
  this.$event('comp.*1') // и передать ему объект события 'comp.*1'
)

// определить обработчик для события 'comp.*1'
document.addEventListener('comp.*1', event => {
  // отобразить первый компонент в компоненте Контент
  content.innerHTML = '<c-component1></c-component1>'
})
```

Вместо полного указания названия события component1 , как мы это делали ранее:

```
// добавить маршрутизатор элементу 'nav'
this.$router(this.$('nav'), null,
  this.$event('component1') // и передать ему объект события 'component1'
)

// определить обработчик для события 'component1'
document.addEventListener('component1', event => {
  // отобразить первый компонент в компоненте Контент
```

```
  content.innerHTML = '<c-component1></c-component1>'
})
```

При создании регулярного выражения в строке, некоторые его специальные символы, как, например, символ `\w`, необходимо предварять дополнительным символом обратной косой черты `\`. Например, событие `\\w+1` ниже, полностью эквивалентно событиям выше:

```
// добавить маршрутизатор элементу 'nav'
this.$router(this.$('nav'), null,
  this.$event('\\w+1') // и передать ему объект события '\\w+1'
)

// определить обработчик для события '\\w+1'
document.addEventListener('\\w+1', event => {
  // отобразить первый компонент в компоненте Контент
  content.innerHTML = '<c-component1></c-component1>'
})
```

Если нам потребуется обработчик для всех событий, то необходимо использовать специальные символы `.*`, как показано ниже:

```
// добавить маршрутизатор элементу 'nav'
this.$router(this.$('nav'), null,
  this.$event('.*') // и передать ему объект события '.*'
)

// определить обработчик для события '.*'
document.addEventListener('.*', event => {
```

```
  // показать название события в консоли
  console.log(event.type)
})
```

Параметр `event` в обработчике события, ссылается на объект события возвращаемый методом `$event()` . Кроме свойства `type` и ряда других полезных свойств, у этого объекта имеется дополнительное свойство `target` , содержащее ссылку, по которой был выполнен щелчок мышкой или объект location, если переход был выполнен не по ссылке, а с помощью кнопок браузера: `Вперёд` или `Назад` .

Каждая ссылка и объект `location` , имеют следующие свойства:

- **href** - полная строка запроса к ресурсу
- **pathname** - путь к ресурсу
- **origin** - общая схема запроса
- **protocol** - протокол
- **port** - порт, используемый ресурсом
- **host** - хост
- **hostname** - название хоста
- **hash** - часть строки, которая идёт после символа решетки (#) в запросе
- **search** - часть строки, которая идёт после знака вопроса (?) в запросе

В этом мы легко можем убедиться, если добавим в обработчик события вывод любого из этих свойств на консоль:

```
// определить обработчик для события 'component1'
document.addEventListener('component1', event => {
  // отобразить первый компонент в компоненте Контент
  content.innerHTML = '<c-component1></c-component1>'

  // вывод свойства 'href' объекта 'target' в консоль браузера
  console.log(event.target.href)
})
```

Используя данные свойства объекта target , можно легко организовать изменение цвета ссылки, маршрут в атрибуте href которой, соответствует текущему событию маршрутизатора.

Давайте сделаем это в нашем примере, для чего внесите изменения в файл menu.htm:

```
<c-menu>
  <nav>
    <a href="/">Главная</a>
    <a href="component1">Компонент 1</a>
    <a href="component2">Компонент 2</a>
  </nav>

  <style>
    a {
      margin-right: 10px;
    }
    .current {
      color: red;
    }
  </style>

  <script>
```

```
// сохранить коллекцию ссылок компонента Меню
const links = this.$$('nav a')

// функция добавления класса 'current' ссылке, соответствующей текущему событию маршрутизатора
function addClass(event) {
  links.forEach(elem => {
    if(event.target.pathname === elem.pathname) elem.classList.add('current')
    else elem.classList.remove('current')
  })
}

// сохранить ссылку на компонент Контент
const content = document.querySelector('c-content')

// определить обработчик для события '/'
document.addEventListener('/', event => {
  // отобразить содержимое компонента Контент по умолчанию
  content.innerHTML = ''

  // вызвать функцию добавления класса
  addClass(event)
})

// определить обработчик для события 'component1'
document.addEventListener('component1', event => {
  // отобразить первый компонент в компоненте Контент
  content.innerHTML = '<c-component1></c-component1>'

  // вызвать функцию добавления класса
  addClass(event)
})

// определить обработчик для события 'component2'
document.addEventListener('component2', event => {
  // отобразить второй компонент в компоненте Контент
```

```
      content.innerHTML = '<c-component2></c-component2>'

      // вызвать функцию добавления класса
      addClass(event)
    })

    // добавить маршрутизатор элементу 'nav' и передать ему объекты событий
    this.$router(this.$('nav'), {
        start: true // проверить события при запуске маршрутизатора
      },
      this.$event('/'),
      this.$event('component1'),
      this.$event('component2')
    )
  </script>
</c-menu>
```

Первое, на что стоит обратить внимание, мы добавили новый класс `current` в стили компонента Меню. Он будет окрашивать в красный цвет ссылку, которая соответствует текущему событию маршрутизатора. Затем мы сохранили в коде коллекцию ссылок в константе `links` и добавили функцию `addClass()`, которая просто перебирает все ссылки маршрутизатора и добавляет класс `current` той, чьё свойство `pathname` соответствует одноимённому свойству объекта `target` из параметра `event`, который передаётся этой функции в первом аргументе. С остальных ссылок маршрутизатора, данный класс удаляется.

Добавление и удаление класса ссылкам, осуществляется с помощью стандартного свойства classList и содержащихся в нём методов. Мы добавили вызов этой функции и передачу ей параметра `event`, в каждый обработчик события маршрутизатора:

```
// определить обработчик для события '/'
document.addEventListener('/', event => {
  // отобразить содержимое компонента Контент по умолчанию
```

```
  content.innerHTML = ''

  // вызвать функцию добавления класса
  addClass(event)
})

// определить обработчик для события 'component1'
document.addEventListener('component1', event => {
  // отобразить первый компонент в компоненте Контент
  content.innerHTML = '<c-component1></c-component1>'

  // вызвать функцию добавления класса
  addClass(event)
})

// определить обработчик для события 'component2'
document.addEventListener('component2', event => {
  // отобразить второй компонент в компоненте Контент
  content.innerHTML = '<c-component2></c-component2>'

  // вызвать функцию добавления класса
  addClass(event)
})
```

Последнее изменение в компоненте Меню, касается уже самого метода $router() . Мы перенесли этот метод в самый конец логики компонента, после определения всех обработчиков его событий:

```
// добавить маршрутизатор элементу 'nav' и передать ему объекты событий
this.$router(this.$('nav'), {
    start: true // проверить события при запуске маршрутизатора
  },
  this.$event('/'),
```

```
    this.$event('component1'),
    this.$event('component2')
  )
```

Это было сделано специально, поскольку в его объекте параметров, мы указали свойство `start` со значением `true`, необходимое для проверки событий при запуске маршрутизатора. А для этого, обработчики событий должны быть определены раньше, чем начнётся проверка событий и произойдёт вызов соответствующего событию обработчика.

После запуска приложения в браузере, первая ссылка станет красного цвета:

Browser

Главная Компонент 1 Компонент 2

Главная

Щелчок по другой ссылке, а также нажатие кнопок: `Вперёд` или `Назад` в браузере, приведёт к окрашиванию в красный цвет ссылки, соответствующей текущему событию маршрутизатора. Цвет остальных ссылок будет сброшен в стандартный, указанный в настройках стилей вашего браузера.

Кроме свойства `start` в объекте параметров метода `$router()`, можно указывать и другие свойства, которые допустимы в последнем аргументе метода addEventListener(). Например:

```
this.$router(this.$('nav'), {
    once: true, // обработчик будет автоматически удалён после выполнения
    passive: true, // обработчик никогда не вызовет метод preventDefault()
    capture: true // событие будет перехвачено при погружении
  },
  this.$event('/'),
  this.$event('component1'),
  this.$event('component2')
)
```

На этом всё! Мы закончили наше изучение библиотеки CompoJS. Но вы сможете узнать о маршрутизаторе больше, если прочитаете руководство по созданию простого приложения.

Приложение

Мы рассмотрим пример создания простого приложения на CompoJS, которое будет выводить список запрошенных с сервера пользователей и при щелчке на любом пользователе из этого списка, будет показывать его данные, тоже загружаемые с сервера. Кроме этого, в целях демонстрации работы маршрутизатора, будет создана дополнительная страница *Описание*, которая будет содержать краткое описание работы нашего приложения.

Приложение будет иметь следующий вид:

Пользователи Список Описание

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

За основу будет взят проект, который мы создали в главе Сборка проекта. Скачайте и разархивируйте файл new.zip. Затем откройте терминал и перейдите из него в разархивированную папку **new**. Введите в терминале команду для создания проекта:

```
npm i
```

В зависимости от вашей операционной системы, возможно вам придётся запускать все команды установки в терминале от имени администратора и вводить пароль. Для этого добавьте в терминале ключевое слово sudo , перед любой командой установки:

```
sudo npm i
```

Наше приложение будет содержать всего пять компонентов. Два компонента в заготовке у нас уже есть и находятся в файлах: component1.htm и component2.htm, расположенные в папке components соответственно.

Переименуйте файл component1.htm в user.htm и внесите в него следующие изменения:

```
<c-user>
  <form>
    <p>
      <label>ID</label>
      <input value="${ user.id }">
    </p>
    <p>
      <label>Имя</label>
      <input value="${ user.firstName }">
    </p>
    <p>
      <label>Фамилия</label>
      <input value="${ user.lastName }">
    </p>
    <a href="/">Назад</a>
  </form>

  <style>
    p {
      margin-bottom: 20px;
```

```
  }
  label {
    font-size: 14px;
    color: gray;
  }
  input {
    display: block;
    margin-top: 5px;
    padding: 10px 15px;
    width: 100%;
    font-size: 14px;
    border: 1px solid #ddd;
    border-radius: 3px;
    box-sizing: border-box;
  }
  a {
    display: inline-block;
    margin-top: 10px;
    padding: 6px 12px;
    color: #fff;
    text-decoration: none;
    font-size: 14px;
    background: #0B77B3;
    border-radius: 3px;
  }
  a:hover {
    background: #0c83c4;
  }
</style>

<script>
  // создать объект пользователя
  this.user = {
    id: '',
    firstName: '',
```

```
      lastName: ''
    }

    // когда компонент Сервер будет определён
    customElements.whenDefined('c-server').then(() => {
      // вызвать обработчик события 'getuser' и передать ему объект события созданный методом '$event()'
      document.dispatchEvent(this.$event('getuser', {
        detail: {
          id: this.dataset.id, // передать 'id' пользователя в обработчик этого события
          user: this.user // передать объект 'user' в обработчик этого события
        }
      }))
    })
  </script>
</c-user>
```

Стилизацию компонентов мы рассматривать не будем, поскольку это выходит за рамки данной темы. Вместо этого, мы сосредоточимся на HTML-содержимом и логике компонента. В качестве HTML-содержимого, наш компонент Пользователь имеет тег <form>. В поля этой формы, будет выводиться вся информация о конкретном пользователе:

```
<form>
  <p>
    <label>ID</label>
    <input value="${ user.id }">
  </p>
  <p>
    <label>Имя</label>
    <input value="${ user.firstName }">
  </p>
  <p>
    <label>Фамилия</label>
    <input value="${ user.lastName }">
```

```
  </p>
  <a href="/">Назад</a>
</form>
```

Наибольший интерес для нас представляет логика этого компонента:

```
<script>
  // создать объект пользователя
  this.user = {
    id: '',
    firstName: '',
    lastName: ''
  }

  // когда компонент Сервер будет определён
  customElements.whenDefined('c-server').then(() => {
    // вызвать обработчик события `getuser` и передать ему объект события созданный методом '$event()'
    document.dispatchEvent(this.$event('getuser', {
      detail: {
        id: this.dataset.id, // передать `id` пользователя в обработчик этого события
        user: this.user // передать объект `user` в обработчик этого события
      }
    }))
  })
</script>
```

Вначале мы создаём объект пользователя user , свойства которого будут выводиться в рассмотренные выше поля формы. После создания этого объекта, мы вызываем метод whenDefined() пользовательских элементов customElements. Методу whenDefined() передаётся строка с названием компонента 'c-server' , после определения которого, данный метод вернёт

промис. В методе then() этого промиса, мы указываем код, который необходимо выполнить после определения компонента <c-server>.

Компонент Сервер мы создадим немного позже. Он будет содержать обработчик события getuser , который мы вызываем методом dispatchEvent() объекта document , внутри метода then() возвращённого промиса. В метод dispatchEvent() мы передаём объект события getuser , возвращаемый методом $event() . В первом аргументе данного метода, ему передаётся строка с названием события, а во втором объект параметров, свойство detail которого, содержит передаваемые в событие данные: id пользователя и объект user .

Передаваемое в компонент Сервер id пользователя для запроса информации о нём из базы данных, будет находиться в атрибуте data-id компонента <c-user>. Доступ к этому атрибуту осуществляется с помощью свойства dataset этого компонента. Давайте создадим второй компонент, который будет выводить содержимое нашего приложения.

Переименуйте файл component2.htm в content.htm и внесите в него изменения показанные ниже:

```
<c-content to="main">
  <!-- Содержимое компонента Контент по умолчанию -->
  <slot>
    <ul c-for="user of users">
      <li>
        <a href="#${ user.id }">${ user.firstName + ' ' + user.lastName }</a>
      </li>
    </ul>
  </slot>

  <style>
    ul {
      margin-bottom: 50px;
      padding-left: 0;
      list-style: none;
```

```
    }
    a {
      display: block;
      margin: 0 0 1px;
      padding: 8px 15px;
      color: #222;
      font-size: 15px;
      text-decoration: none;
      background: #fafafa;
      border: 1px solid #ddd;
    }
    a:hover {
      color: #fff;
      background: #0B77B3;
    }
    li:not(:last-child) a {
      border-bottom: none;
    }
  </style>

  <script>
    // создать пустой массив объектов пользователей
    this.users = []

    // когда компонент Сервер будет определён
    customElements.whenDefined('c-server').then(() => {
      // вызвать обработчик события 'getusers' и передать ему объект события созданный методом '$event()'
      document.dispatchEvent(this.$event('getusers', {
        detail: this // передать компонент Контент в обработчик этого события
      }))
    })
  </script>
</c-content>
```

Компонент Контент мы будем подключать в стандартный тег <main>, главного файла **index.html** нашего приложения. Для этого указывается атрибут `to` со значением `main` , в открывающем теге <c-content> данного компонента. HTML компонента Контент содержит безымянный слот, который будет выводить содержимое этого компонента по умолчанию, т.е. когда тег монтирования компонента Контент, не будет иметь передаваемого в слот HTML-содержимого.

Содержимое по умолчанию будет выводиться в цикле и представлять собой список пользователей запрошенных с сервера, и помещённых в массив `users` :

```
<!-- Содержимое компонента Контент по умолчанию -->
<slot>
  <ul c-for="user of users">
    <li>
      <a href="#${ user.id }">${ user.firstName + ' ' + user.lastName }</a>
    </li>
  </ul>
</slot>
```

После создания этого массива и определения компонента <c-server>, мы вызываем в методе `then()` обработчик события `getusers` :

```
<script>
  // создать пустой массив объектов пользователей
  this.users = []

  // когда компонент Сервер будет определён
  customElements.whenDefined('c-server').then(() => {
    // вызвать обработчик события 'getusers' и передать ему объект события созданный методом '$event()'
    document.dispatchEvent(this.$event('getusers', {
      detail: this // передать компонент Контент в обработчик этого события
```

```
    }))
  })
</script>
```

В свойстве `detail` объекта события `getusers`, его обработчику передаётся весь компонент Контент целиком, а точнее, ссылка на его свойство `$data`, которому соответствует ключевое слово `this` в логике компонента.

Через этот объект данных, можно получить доступ к любым свойствам компонента и изменить их значения. В нашем случае, нам потребуется доступ к пользовательскому свойству `users`. Когда информация из базы данных будет получена в обработчике `getusers`, то она будет преобразована в массив, и новый массив будет присвоен этому свойству.

Давайте напишем компонент Сервер. Для этого создайте в папке components новый файл server.htm:

```
<c-server>
  <script>
    /* создать обработчик для события 'getusers' и при наступлении этого события,
    получить с сервера массив объектов всех пользователей */
    document.addEventListener('getusers', event => {
      fetch('https://rem-rest-api.herokuapp.com/api/users?limit=100', {
        method: 'GET' // метод запроса
      })
      // преобразовать полученный ответ в JSON
      .then(response => response.json())
      // присвоить свойству 'users' компонента Контент, полученный массив объектов пользователей
      .then(result => event.detail.users = result.data)
    })

    /* создать обработчик для события 'getuser' и при наступлении этого события,
    получить с сервера объект конкретного пользователя */
    document.addEventListener('getuser', event => {
```

```
      // добавить к запросу идентификатор пользователя
      fetch('https://rem-rest-api.herokuapp.com/api/users/' + event.detail.id, {
        method: 'GET' // метод запроса
      })
      // преобразовать полученный ответ в JSON
      .then(response => response.json())
      .then(result => {
        // добавить в объект 'user' компонента Пользователь, полученный идентификатор пользователя
        event.detail.user.id = result.id
        // добавить в объект 'user' компонента Пользователь, полученное имя пользователя
        event.detail.user.firstName = result.firstName
        // добавить в объект 'user' компонента Пользователь, полученную фамилию пользователя
        event.detail.user.lastName = result.lastName
      })
    })
  </script>
</c-server>
```

Компонент Сервер не имеет никакого HTML и содержит два обработчика, которые мы вызывали в компонентах: Пользователь и Контент. Рассмотрим вначале первый обработчик события getusers , который вызывался из компонента Контент:

```
/* создать обработчик для события 'getusers' и при наступлении этого события,
получить с сервера массив объектов всех пользователей */
document.addEventListener('getusers', event => {
  fetch('https://rem-rest-api.herokuapp.com/api/users?limit=100', {
    method: 'GET' // метод запроса
  })
  // преобразовать полученный ответ в JSON
  .then(response => response.json())
  // присвоить свойству 'users' компонента Контент, полученный массив объектов пользователей
```

```
    .then(result => event.detail.users = result.data)
})
```

С помощью функции fetch() мы делаем запрос на тестовый сервер rem-rest-api.herokuapp.com, который содержит базу данных с нашими пользователями. Когда сервер вернёт ответ, то он будет преобразован в формат JSON и получившийся в итоге объект, будет содержать свойство `data`, в котором будет находиться массив объектов пользователей. Этот массив объектов мы присваиваем свойству `users` компонента Контент, доступ к которому мы получаем через свойство `detail` параметра `event` обработчика события `getusers`.

Второй обработчик события называется `getuser` и он вызывался из компонента Пользователь:

```
/* создать обработчик для события 'getuser' и при наступлении этого события,
получить с сервера объект конкретного пользователя */
document.addEventListener('getuser', event => {
  // добавить к запросу идентификатор пользователя
  fetch('https://rem-rest-api.herokuapp.com/api/users/' + event.detail.id, {
    method: 'GET' // метод запроса
  })
  // преобразовать полученный ответ в JSON
  .then(response => response.json())
  .then(result => {
    // добавить в объект 'user' компонента Пользователь, полученный идентификатор пользователя
    event.detail.user.id = result.id
    // добавить в объект 'user' компонента Пользователь, полученное имя пользователя
    event.detail.user.firstName = result.firstName
    // добавить в объект 'user' компонента Пользователь, полученную фамилию пользователя
    event.detail.user.lastName = result.lastName
  })
})
```

Он просто получает с сервера данные конкретного пользователя, преобразует их в JSON и присваивает значения свойствам объекта user , который мы передавали в этот обработчик из компонента Пользователь:

```
// вызвать обработчик события 'getuser' и передать ему объект события созданный методом '$event()'
document.dispatchEvent(this.$event('getuser', {
  detail: {
    id: this.dataset.id, // передать 'id' пользователя в обработчик этого события
    user: this.user // передать объект 'user' в обработчик этого события
  }
}))
```

Как видно из листинга выше, что кроме объекта user , мы ещё передавали в обработчик свойство id из атрибута data-id , доступ к которому осуществляется через свойство dataset компонента Пользователь. Значение этого id мы указываем в обработчике getuser , при запросе пользователя с сервера по его идентификатору:

```
// добавить к запросу идентификатор пользователя
fetch('https://rem-rest-api.herokuapp.com/api/users/' + event.detail.id, {
  method: 'GET' // метод запроса
})
```

Доступ к id пользователя в обработчике, реализуется через свойство detail параметра event . Но как мы помним, в самом компоненте Пользователь данное свойство отсутствует. Оно берётся из его атрибута data-id , который добавляется компоненту Пользователь, при создании этого компонента в маршрутизаторе компонента Меню.

Создайте в папке components новый файл menu.htm:

```
<c-menu>
  <nav>
    <a class="logo" href="/">Пользователи</a>
    <a href="/">Список</a>
    <a href="descript">Описание</a>
  </nav>

  <style>
    nav {
      display: flex;
      margin-top: 10px;
      margin-bottom: 30px;
    }
    a {
      margin-right: 10px;
      color: rgb(62, 99, 168);
      text-decoration: none;
      font-size: 16px;
    }
    a:hover:not(.logo):not(.current) {
      opacity: .8;
    }
    .current {
      color: red;
    }
    .logo {
      margin-right: auto;
      color: rgb(112, 112, 112);
    }
  </style>

  <script>
    // сохранить коллекцию ссылок компонента Меню
    const links = this.$$('nav a')
```

```
// функция добавления класса 'current' ссылке, соответствующей текущему событию маршрутизатора
function addClass(event) {
  links.forEach(elem => {
    if(event.target.pathname === elem.pathname) elem.classList.add('current')
    else elem.classList.remove('current')
  })
}

// функция удаления класса 'current' у всех ссылок
function removeClass(event) {
  links.forEach(elem => elem.classList.remove('current'))
}

// сохранить ссылку на компонент Контент
const content = document.querySelector('[is="c-content"]')

// определить обработчик для события '/'
document.addEventListener('/', event => {
  // отобразить содержимое компонента Контент по умолчанию
  content.innerHTML = ''

  // вызвать функцию добавления класса
  addClass(event)
})

// определить обработчик для события 'descript'
document.addEventListener('descript', event => {
  // отобразить компонент Описание в компоненте Контент
  content.innerHTML = '<c-descript></c-descript>'

  // вызвать функцию добавления класса
  addClass(event)
})

// определить обработчик для события 'id' пользователей
```

```
    document.addEventListener('/#\\d+', event => {
      /* отобразить компонент Пользователь в компоненте Контент и
      передать ему идентификатор пользователя в атрибуте 'data-id' */
      content.innerHTML = `<c-user data-id="${event.target.hash.substring(1)}"></c-user>`

      // вызвать функцию удаления класса
      removeClass()
    })

    // добавить маршрутизатор для всего документа и передать ему объекты событий
    this.$router(document, {
        start: true // проверить события при запуске маршрутизатора
      },
      this.$event('/'),
      this.$event('descript'),
      this.$event('/#\\d+')
    )
  </script>
</c-menu>
```

Компонент Меню содержит тег <nav> с тремя ссылками. Маршруты в атрибутах `href` первых двух ссылок совпадают и ведут на главную страницу, которая называется *Список*. Последняя ссылка ведёт на страницу *Описание*, чей компонент мы создадим немного позже. Теперь давайте разберём подробнее, что происходит в логике компонента Меню.

Сначала мы сохраняем статическую коллекцию его ссылок в константе `links`:

```
// сохранить коллекцию ссылок компонента Меню
const links = this.$$('nav a')
```

Затем мы определяем две вспомогательные функции. Первая для добавления класса `current` той ссылке, значение атрибута `href` которой соответствует текущему событию маршрутизатора. А вторая функция, просто удаляет данный класс у всех ссылок:

```
// функция добавления класса 'current' ссылке, соответствующей текущему событию маршрутизатора
function addClass(event) {
  links.forEach(elem => {
    if(event.target.pathname === elem.pathname) elem.classList.add('current')
    else elem.classList.remove('current')
  })
}

// функция удаления класса 'current' у всех ссылок
function removeClass(event) {
  links.forEach(elem => elem.classList.remove('current'))
}
```

Потом мы сохраняем в константе `content` ссылку на тег монтирования компонента Контент, чтобы к нему было проще обращаться в обработчиках:

```
// сохранить ссылку на компонент Контент
const content = document.querySelector('[is="c-content"]')
```

После этого у нас идёт определение трёх обработчиков, первый из которых соответствует событию `/` :

```
// определить обработчик для события '/'
document.addEventListener('/', event => {
```

```
  // отобразить содержимое компонента Контент по умолчанию
  content.innerHTML = ''

  // вызвать функцию добавления класса
  addClass(event)
})
```

В обработчике этого события отображается содержимое компонента Контент по умолчанию, т.е. когда ему не передаётся никакой внешний HTML. Это происходит при загрузке приложения в браузер или достигается с помощью очистки свойства innerHTML , тега монтирования данного компонента.

Обработчик будет вызван в тот момент, когда пользователь выполнит щелчок по ссылке меню, чей атрибут href содержит значение / , и после нажатия кнопок: Вперёд или Назад в браузере, когда свойство href глобального объекта location, будет иметь такое же значение.

В конце обработчика мы вызываем функцию addClass() , которая просто перебирает все ссылки маршрутизатора и добавляет класс current той, чьё свойство pathname соответствует одноимённому свойству объекта target параметра event , который передаётся этой функции в первом её аргументе:

```
// функция добавления класса 'current' ссылке, соответствующей текущему событию маршрутизатора
function addClass(event) {
  links.forEach(elem => {
    if(event.target.pathname === elem.pathname) elem.classList.add('current')
    else elem.classList.remove('current')
  })
}
```

Параметр event содержит свойство target , которое ссылается на объект инициировавший данное событие, например, ссылка меню или объект location . С остальных ссылок маршрутизатора, данный класс удаляется.

Второй обработчик вызывается для события descript и напоминает первый:

```
// определить обработчик для события 'descript'
document.addEventListener('descript', event => {
  // отобразить компонент Описание в компоненте Контент
  content.innerHTML = '<c-descript></c-descript>'

  // вызвать функцию добавления класса
  addClass(event)
})
```

При его срабатывании, свойству innerHTML тега монтирования компонента Контент, присваивается тег монтирования компонента Описание. Под присваиванием, здесь подразумевается присвоение этому свойству простой строки в кавычках, содержащей тег монтирования компонента. Можно использовать и самозакрывающиеся теги в строке, например: <c-descript/>.

Последний обработчик соответствует событию id пользователей:

```
// определить обработчик для события 'id' пользователей
document.addEventListener('/#\\d+', event => {
  /* отобразить компонент Пользователь в компоненте Контент и
  передать ему идентификатор пользователя в атрибуте 'data-id' */
  content.innerHTML = `<c-user data-id="${event.target.hash.substring(1)}"></c-user>`

  // вызвать функцию удаления класса
```

```
  removeClass()
})
```

В качестве этого события, выступает регулярное выражение вида: `/#\\d+`. Обработчик будет вызван для всех значений свойства `href` ссылок меню или объекта `location`, которые начинаются с символа решётка `#` и последующими за ним цифрами, являющимися идентификаторами пользователей. Мы поместили тег монтирования компонента Пользователь в шаблонную строку:

```
content.innerHTML = `<c-user data-id="${event.target.hash.substring(1)}"></c-user>`
```

В этой строке мы создаём для компонента Пользователь атрибут `data-id` и передаём ему значение из свойства `hash` целевого объекта `target`, которым может выступать ссылка меню или объект `location`. Нам нужны только цифры из хеша целевого объекта, являющиеся идентификатором пользователя, без самого символа решётки. Поэтому в конце используется метод substring() с аргументом `1`, который просто возвращает подстроку начиная с первого символа.

Доступ к атрибуту `data-id` в компоненте Пользователь, осуществляется с помощью свойства `dataset`:

```
// вызвать обработчик события 'getuser' и передать ему объект события созданный методом '$event()'
document.dispatchEvent(this.$event('getuser', {
  detail: {
    id: this.dataset.id, // передать 'id' пользователя в обработчик этого события
    user: this.user // передать объект 'user' в обработчик этого события
  }
}))
```

В конце обработчика вызывается функция removeClass() , которая просто удаляет класс current у всех ссылок меню:

```
// вызвать функцию удаления класса
removeClass()
```

После определения обработчиков событий, мы создаём маршрутизатор для всего объекта документа приложения:

```
// добавить маршрутизатор для всего документа и передать ему объекты событий
this.$router(document, {
    start: true // проверить события при запуске маршрутизатора
  },
  this.$event('/'),
  this.$event('descript'),
  this.$event('/#\\d+')
)
```

Это необходимо для того, чтобы обрабатывать события не только из компонента Меню, но ещё из компонента Контент, который содержит в качестве содержимого по умолчанию, список ссылок с нашими пользователями:

```
<!-- Содержимое компонента Контент по умолчанию -->
<slot>
  <ul c-for="user of users">
    <li>
      <a href="#${ user.id }">${ user.firstName + ' ' + user.lastName }</a>
    </li>
  </ul>
</slot>
```

Если бы мы добавили маршрутизатор только к тегу <nav> компонента Меню, то он не реагировал бы на события ссылок из компонента Контент и не выводил бы данные конкретного пользователя, при щелчке по любой из ссылок этого списка. Поэтому необходимо добавить маршрутизатор для объекта `document`, на который всплывают события из всех открытых компонетов приложения.

В объекте параметров маршрутизатора, мы указываем свойство `start` со значением `true`. Это обеспечит проверку всех событий при его запуске и вызов соответствующих обработчиков. Поэтому прежде были определены все обработчики, и только потом был вызван маршрутизатор. Тем самым, мы гарантируем добавление класса `current` той ссылке, которая будет соответствовать текущему событию маршрутизатора при запуске приложения.

Вызов маршрутизатора без параметра `start` со значением `true`, можно располагать в любом месте логики компонента. Давайте добавим последний компонент Описание.

Создайте в папке **components** новый файл **descript.htm**:

```
<c-descript>
  <h1>Описание</h1>
  <p>Это простое приложение написанное на <a href="https://github.com/compo-js/components" target="_blank">CompoJS</a
    делает запрос к <a href="https://rem-rest-api.herokuapp.com" target="_blank">тестовому</a> серверу и выводит спи
    При нажатии на пользователя в этом списке, открывается страница с его данными.</p>

  <style>
    h1 {
      font-size: 24px;
    }
    p {
      line-height: 1.5;
    }
    a {
      color: rgb(62, 99, 168);
```

```
    }
  </style>
</c-descript>
```

Это очень простой компонент и, как следует из его названия, он просто выводит краткое описание того, что делает наше приложение. Данный компонент был добавлен для более развёрнутого примера работы с маршрутизатором.

В целях демонстрации работы со статическими файлами, давайте добавим ещё и внешнюю таблицу стилей в приложение. Создайте в папке **assets** файл **style.css**:

```
@import 'https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.1/normalize.min.css';

body {
  padding: 10px;
  font-family: sans-serif;
}
```

В начале файла стилей идёт подключение normalize.css, необходимое для нормализации браузерных стилей нашего приложения. Затем мы задаём тегу <body> два правила: небольшие внутренние отступы и шрифт без засечек.

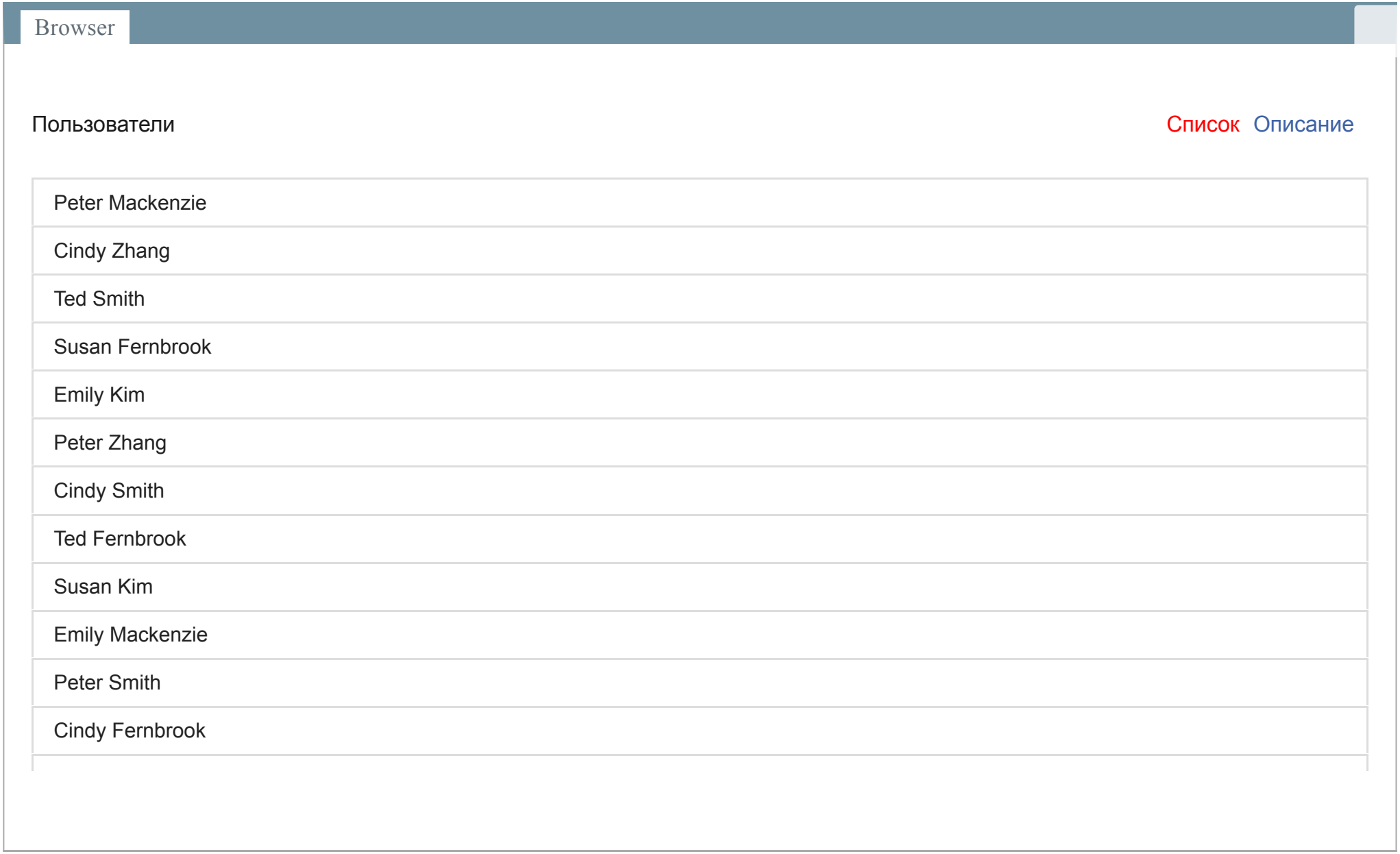
Нам осталось только внести изменения в файл **index.html**:

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Пользователи</title>
  <link rel="stylesheet" href="dist/style.css">
</head>
<body>
  <!-- подключить компонент Меню -->
  <c-menu></c-menu>

  <!-- подключить компонент Контент в тег 'main' -->
  <main is="c-content"></main>

  <!-- подключить компонент Сервер -->
  <c-server></c-server>

  <!-- тег подключения файла компонентов -->
  <template src="dist/components.htm"></template>

  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

В теге <head> мы подключаем нашу таблицу стилей, а в теге <body>, сначала мы подключаем компонент Меню, затем монтируем компонент Контент в тег <main> и в конце, мы подключаем компонент Сервер.

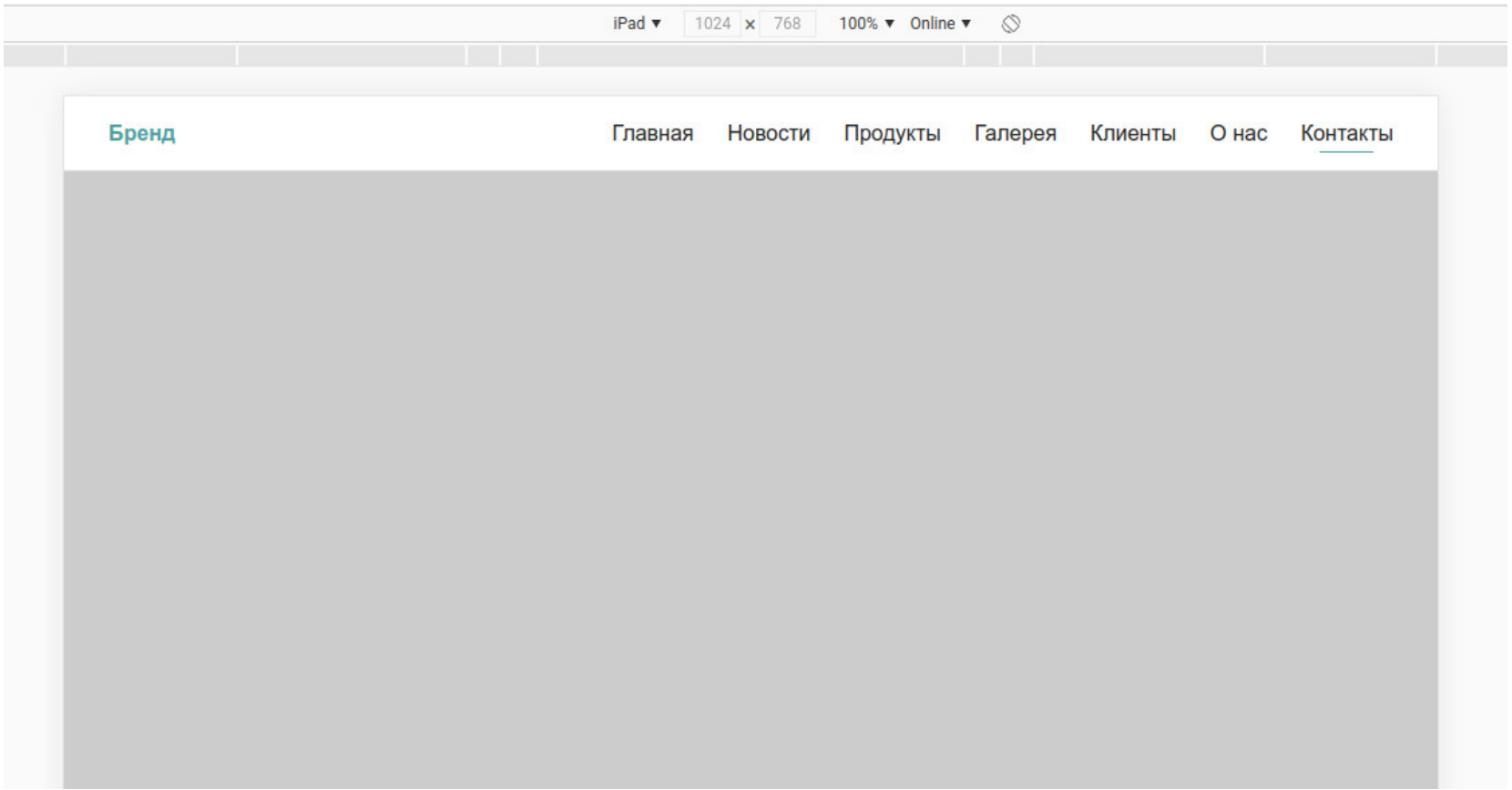
Откройте наше приложение в браузере и на экране вы увидите:

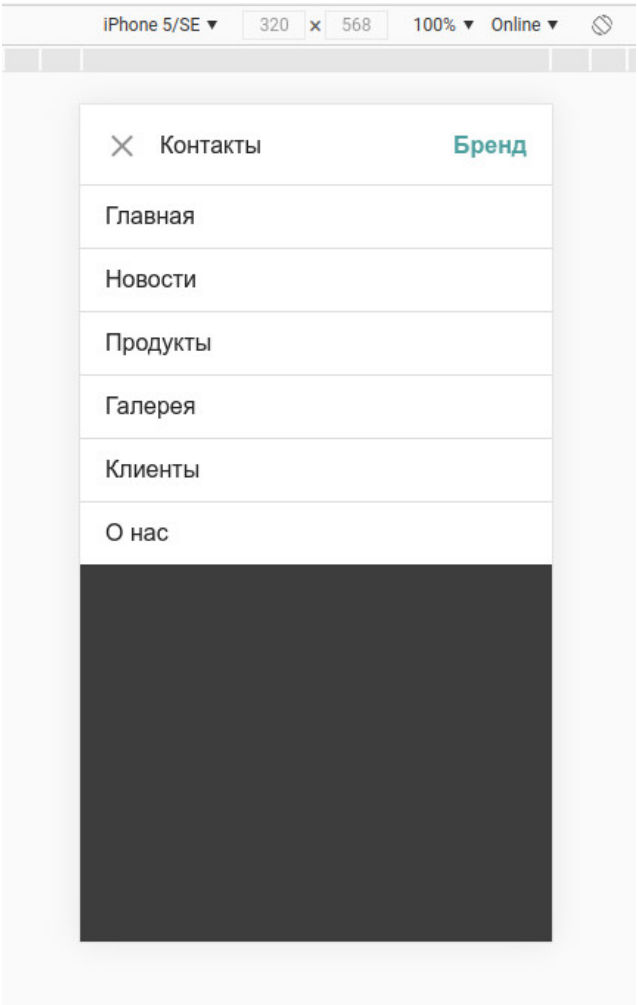


Скачать приложение вы можете по ссылке userlist.zip. Мы рассмотрели простой пример создания приложения с помощью CompoJS. Используя полученную практику, вы сможете создавать собственные приложения различного уровня сложности.

Пример

Здесь представлен наглядный пример создания компонента одноуровневого Меню в CompoJS, с подробным описанием в комментариях. Вы можете создавать собственные коллекции компонентов и выкладывать их для общедоступного использования.





Файл **index.html**

```html
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Меню</title>
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.1/normalize.min.css">
  <style>
    /* добавить стили тегу 'body', чтобы продемонстрировать
      работу компонента 'c-menu' в окне браузера */
    body {
      min-height: 1000px;
      background: #ccc;
    }
  </style>
</head>
<body>
  <!-- монтировать компонент 'c-menu' в тег 'nav' и добавить этому тегу атрибут 'hidden',
    чтобы нестилизованные элементы компонента не отображались, пока компонент не будет полностью готов -->
  <nav is="c-menu" hidden>
    <!-- импортировать Логотип в компонент 'c-menu', через именованный слот 'logo' -->
    <a href="/" slot="logo">Бренд</a>

    <!-- импортировать пункты Меню в компонент 'c-menu', через именованный слот 'item' -->
    <a href="/" slot="item">Главная</a>
    <a href="#news" slot="item">Новости</a>
    <a href="#products" slot="item">Продукты</a>
    <a href="#gallery" slot="item">Галерея</a>
    <a href="#clients" slot="item">Клиенты</a>
    <a href="#about" slot="item">О нас</a>
    <a href="#contacts" slot="item">Контакты</a>
  </nav>

  <!-- подключить файл компонентов -->
  <template src="dist/components.htm"></template>
```

```
  <!-- подключить CompoJS -->
  <script src="dist/compo.min.js"></script>
</body>
</html>
```

Файл menu.htm

```
<c-menu to="nav">
  <!-- HTML-содержимое компонента -->
  <div class="wrapper">
    <div class="head">
      <button class="button"></button>
      <a href="#" class="page"></a>
      <!-- в этот слот будет монтироваться Логотип -->
      <slot name="logo"></slot>
    </div>
    <div class="list">
      <!-- в этот слот будут монтироваться пункты Меню -->
      <slot name="item"></slot>
    </div>
  </div>


  <!-- Стили компонента -->
  <style>
    /* CSS-переменные компонента, значения которым можно задать
      через атрибуты data-* его тега монтирования, например:

      <nav is="c-menu" hidden
        data-height-menu="70px"
        data-font-size="1.5em"
```

```
    data-opacity=".9"
    data-duration="600ms"
  >
    <!-- импортировать Логотип в компонент 'c-menu', через именованный слот 'logo' -->
    <a href="/#" slot="logo">Бренд</a>
    ...
  </nav>

  Если атрибуты data-* не используются, то этим переменным устанавливаются значения по умолчанию.
  В примере ниже, значением по умолчанию для перепенной --height-menu будет '55px'

  --height-menu: ${ this.dataset.heightMenu || '55px' };
*/
:host {
  --height-menu: ${ this.dataset.heightMenu || '55px' };
  --font-size: ${ this.dataset.fontSize || '1em' };
  --main-color: ${ this.dataset.mainColor || '#222' };
  --second-color: ${ this.dataset.secondColor || 'cadetblue' };
  --background-color: ${ this.dataset.backgroundColor || '#fff' };
  --border-color: ${ this.dataset.borderColor || '#ddd' };
  --border-width: ${ this.dataset.borderWidth || '1px' };
  --opacity: ${ this.dataset.opacity || '.5' };
  --duration: ${ this.dataset.duration || '200ms' };
  --small: ${ small };
  --medium: ${ medium };
  --large: ${ large };
}

/* Общие стили для всех элементов компонента, кроме находящихся в слотах */
* {
  box-sizing: border-box;
  -webkit-tap-highlight-color: rgba(0, 0, 0, 0);
}

/* Стили для элемента-хозяина (тег монтирования) компонента  */
```

```
:host {
  position: fixed;
  z-index: 10000;
  width: 100%;
  top: 0;
  height: var(--height-menu);
  overflow: hidden;
  font-family: sans-serif;
  border-bottom: var(--border-width) solid var(--border-color);
  background: rgba(0,0,0,.7);
  transition: height var(--duration);
}
:host(.host--open) {
  height: 100%;
  border-bottom: none;
}

/* Стили для ссылки Логотипа из светлого DOM, передаваемой в слот с именем 'logo' компонента.
  Подробнее: https://learn.javascript.ru/shadow-dom-style#primenenie-stiley-k-soderzhimomu-slotov
*/
::slotted(a[slot="logo"]) {
  margin-right: 14px;
  padding: 3px;
  font-weight: 600;
  font-size: var(--font-size);
  text-decoration: none;
  color: var(--second-color);
}

/* Стили для ссылок Меню из светлого DOM, передаваемых в слот с именем 'item' компонента.
  Подробнее: https://learn.javascript.ru/shadow-dom-style#primenenie-stiley-k-soderzhimomu-slotov
*/
::slotted(a[slot="item"]) {
  display: block;
  padding: 12px 17px;
```

```
  font-size: var(--font-size);
  text-decoration: none;
  color: var(--main-color);
  border-top: var(--border-width) solid var(--border-color);
  background-color: var(--background-color) !important;
}
::slotted(a[slot="item"]:hover) {
  color: #fff;
  background: var(--main-color) !important;
}
::slotted(a[slot="item"].item--hide) {
  display: none;
}


/* ------  Стили для остальных элементов компонента ------ */
.wrapper {
  height: 100%;
}

.head {
  height: var(--height-menu);
  display: flex;
  align-items: center;
  background: var(--background-color);
}

.button {
  position: relative;
  width: 22px;
  height: 14px;
  margin-left: 17px;
  border: none;
  outline: none;
  cursor: pointer;
```

```
  background: transparent;
  border-top: 2px solid var(--main-color);
  border-bottom: 2px solid var(--main-color);
  transition: all var(--duration);
}
.button:hover {
  opacity: var(--opacity);
}
.button::before,
.button::after {
  content: '';
  position: absolute;
  height: 2px;
  width: 100%;
  left: 0;
  top: 50%;
  margin-top: -1px;
  background: var(--main-color);
  transition: transform var(--duration);
}
.button--close {
  border-top-color: transparent;
  border-bottom-color: transparent;
}
.button--close::before,
.button--close::after {
  width: calc(100% - 3px);
  left: 2px;
}
.button--close::before {
  transform: rotate(45deg);
}
.button--close::after {
  transform: rotate(-45deg);
}
```

```
.page {
  text-decoration: none;
  margin-left: 12px;
  margin-right: auto;
  padding: 3px;
  font-size: var(--font-size);
  color: var(--main-color);
  transition: opacity var(--duration);
}
.page:hover {
  opacity: var(--opacity);
}

.list {
  height: calc(100% - var(--height-menu) + var(--border-width));
  margin-top: calc(-1 * var(--border-width));
}
.list--overflow-auto {
  overflow-y: auto;
}
.list--overflow-hide {
  overflow: hidden;
}


/* Медиазапрос со стилями компонента. В медиазапросы нельзя передавать CSS-переменные, например:

  @media (min-width: var(--medium)) {
    ...
  }

  Этот код не сработает! Но можно использовать подстановки из CompoJS, что и показано ниже.
*/
@media (min-width: ${ medium }) {
```

```
:host {
  overflow: visible;
  background: var(--background-color);
}
:host(.host--open) {
  height: auto;
  border-bottom: var(--border-width) solid var(--border-color);
}

::slotted(a[slot="logo"]) {
  margin-right: 0;
  margin-left: 14px;
}

::slotted(a[slot="item"]) {
  display: inline;
  margin-right: 5px;
  padding: 5px 10px;
  border-top: none;
  background: none !important;
  transition: opacity var(--duration);
}
::slotted(a[slot="item"]:last-of-type) {
  margin-right: 0;
}
::slotted(a[slot="item"]:hover) {
  color: var(--main-color);
  opacity: var(--opacity);
  background: none !important;
}
::slotted(a[slot="item"].item--hide) {
  display: inline-block;
}
::slotted(a[slot="item"].item--active) {
  position: relative;
```

```
}
::slotted(a[slot="item"].item--active)::after {
  content: '';
  position: absolute;
  left: 50%;
  bottom: 0;
  width: 40px;
  height: 1px;
  margin-left: -20px;
  background: var(--second-color);
}

.wrapper {
  display: flex;
  align-items: center;
  max-width: var(--medium);
  margin: 0 auto;
}

.head {
  margin-right: auto;
  background: none;
}

.button {
  display: none;
}

.page {
  display: none;
}

.list {
  display: inline;
  height: auto;
```

```
      margin-top: 0;
      margin-right: 7px;
    }
  }
</style>


<!-- Логика компонента -->
<script>
  const host = this.$root.host
  const button = this.$('.button')
  const page = this.$('.page')
  const list = this.$('.list')
  const logo = this.$('slot[name="logo"]').assignedElements()[0]
  const links = this.$('slot[name="item"]').assignedElements()
  let isClick = false

  /* Определить пользовательские свойства компонента. Задать значения этим свойствам, вы можете из атрибутов
    data-* тега монтирования компонента, как и для CSS-переменных показанных выше. Если атрибуты data-* не использу
    то свойствам устанавливаются значения по умолчанию. Например, свойству small устанавливается значение по умолча
  */
  this.small = this.dataset.small || '600px'
  this.medium = this.dataset.medium || '992px'
  this.large = this.dataset.large || '1200px'

  function pageSetting(target = location) {
    links.forEach(link => {
      if(link.href === target.href) {
        link.classList.add('item--hide')
        link.classList.add('item--active')
        page.textContent = link.textContent
        page.href = link.href
      }
      else {
        link.classList.remove('item--hide')
```

```
      link.classList.remove('item--active')
    }
  })
}

function toggleOverflow(toggle) {
  setTimeout(() => {
    list.classList[toggle]('list--overflow-auto')
    list.scrollTop = 0
    /* Чтобы получить доступ к CSS-переменным в JavaScript, используется метод getComputedStyle() объекта window
      Подробнее: https://developer.mozilla.org/ru/docs/Web/API/Window/getComputedStyle
    */
  }, parseInt(window.getComputedStyle(host).getPropertyValue('--duration')))
}

function clickHandler(e) {
  isClick = true
  if(host.classList.contains('host--open')) button.click()
  pageSetting(e.target)
  setTimeout(() => isClick = false, 0)
}

pageSetting()

button.addEventListener('click', () => {
  if(document.documentElement.clientWidth >= parseInt(this.medium)) return
  host.classList.toggle('host--open')

  if(host.classList.contains('host--open')) {
    button.classList.add('button--close')
    list.classList.remove('list--overflow-hide')
    toggleOverflow('add')
  }
  else {
    button.classList.remove('button--close')
```

```
        list.classList.add('list--overflow-hide')
        toggleOverflow('remove')
      }
    })

    page.addEventListener('click', clickHandler)
    logo.addEventListener('click', clickHandler)
    list.addEventListener('click', clickHandler)

    window.addEventListener('popstate', function() {
      if(isClick) return isClick = false
      if(host.classList.contains('host--open')) button.click()
      pageSetting()
    })

    window.addEventListener('keydown', function(e) {
      if(e.which == 27 && host.classList.contains('host--open')) button.click()
    })

    // Делаем тег монтирования компонента снова видимым
    this.$root.host.removeAttribute('hidden')
  </script>
</c-menu>
```

Скачать компонент Меню, вы можете по ссылке menu.zip.

$root

Свойство `$root` предоставляет прямой доступ к теневому DOM компонента.

```
document.querySelector('my-component').$root.innerHTML = 'Новое содержимое компонента'
```

$data

Свойство `$data` предоставляет прямой доступ к объекту данных компонента.

```
document.querySelector('my-component').$data.myProp = 'Новое значение свойства'
```

$(selector)

Метод `$()` позволяет выбрать один элемент по указанному селектору из HTML-содержимого компонента.

```
document.querySelector('my-component').$('h1').innerHTML = 'Новое содержимое элемента'
```

$$(selector)

Метод $$() позволяет выбрать все элементы по указанному селектору из HTML-содержимого компонента.

```
document.querySelector('my-component').$$('h1')[0].innerHTML = 'Новое содержимое элемента'
```

$connected(function A, function B, ...function N)

Метод $connected() получает любое количество функций, которые будут вызваны при добавлении компонента в документ.

```
this.$connected(
  () => console.log('Компонент добавлен в документ!'),
  function() {
    console.log('Компонент уже добавлен.')
  }
)
```

$disconnected(function A, function B, ...function N)

Метод $disconnected() получает любое количество функций, которые будут вызваны при удалении компонента из документа.

```
this.$disconnected(
  () => console.log('Компонент удалён из документа!'),
  function() {
    console.log('Компонент уже удалён.')
  }
)
```

$adopted(function A, function B, ...function N)

Метод `$adopted()` получает любое количество функций, которые будут вызваны при перемещении компонента в новый документ.

```
this.$adopted(
  () => console.log('Компонент перемещён из документа!'),
  function() {
    console.log('Компонент уже перемещён.')
  }
)
```

$event(name, properties)

Метод `$event()` создаёт и возвращает новый объект событий. Он получает в первом аргументе строку с названием события, а во втором объект с параметрами конструктора CustomEvent .

```
const myEvent = this.$event('myevent', {
  detail: myData
})
```

$router(element, properties, $event A, $event B, ...$event N)

Метод `$router()` создаёт новый маршрутизатор. Он получает в первом аргументе HTML-элемент, а во втором объект с параметрами метода addEventListener() или `null`. Все его последующие аргументы, являются объектами событий возвращаемые методом $event().

```
this.$router(this.$('nav'), null,
  this.$event('/'),
  this.$event('component1'),
  this.$event('component2')
)
```