Урок 5

Фреймворк Vue.js

Основы разработки одностраничных приложений с помощью фреймворка Vue.js.

Для ускорения разработки приложений используются фреймворки. Это готовые наборы решений самых популярных задач. Благодаря фреймворкам разработчику не нужно думать о том, как организовать структуру приложения. У него уже есть все основные инструменты для рутинных задач: обработки событий, разделения приложения на компоненты и связывания компонентов между собой.

На конец 2018 года самые популярные JavaScript-фреймворки – Angular от Google, React от Facebook и Vue.js от независимого разработчика Эвана Ю. У каждого из этих фреймворков есть плюсы и минусы, но в целом все они подходят для создания приложений. Мы подробно остановимся на Vue.js.

Сотрудник Google Эван Ю выпустил первую версию Vue.js в октябре 2015 года. По его задумке, фреймворк должен был стать инструментом для быстрого прототипирования сложных интерфейсов в противовес сложному Angular и сырому в то время React. Фреймворк быстро набрал популярность. Сейчас он поддерживается компаниями и сообществами. Эван Ю получает 16 тысяч долларов в месяц в качестве пожертвований на Patreon.

Установка

Vue.js работает во всех браузерах, начиная с Internet Explorer 8. На официальном сайте проекта можно скачать одну из двух версий: для разработки и для использования в готовом приложении. Обе работают одинаково, разница в том, что в версии для разработки удобный для чтения исходный код и служебные оповещения, которые помогают отлаживать приложение.

Чтобы не скачивать исходники, подключим **Vue.js** через CDN: сам файл будет находиться на удалённом сервере, а мы подключим его к проекту как обычный внешний скрипт. Создадим файл **index.html** и добавим в него тег **<script>** со ссылкой на файл:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Vue.js</title>
</head>
<body>
  <script src="https://cdn.jsdelivr.net/npm/vue@2.5.17/dist/vue.js"></script>
</body>
</html>
```

Теперь добавим отдельно файл **script.js**, в котором будем писать наше приложение. Подключим этот файл в **index.html**:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Vue.js</title>
</head>
<body>
  <script src="https://cdn.jsdelivr.net/npm/vue@2.5.17/dist/vue.js"></script>
  <script src="script.js"></script>
</body>
</html>
```

Теперь проверим, что **Vue.js** успешно подключён. Попробуем создать экземпляр класса **Vue**:

```
//script.js
const app = new Vue();
console.log(app);
```

Если всё работает верно, в консоли появился объект класса **Vue**:

```
Vue {_uid: 0, _isVue: true, $options: {...}, _renderProxy: Proxy, _self: Vue, ...}
```

Теперь библиотека работает, и можно начинать создавать приложение.

Шаблоны

Прежде всего нужно научиться подставлять данные в html-шаблон. Чтобы указать, какая именно часть нашего html должна восприниматься Vue как шаблон, добавим блок с **id="app"** и привяжем его к объекту **app**. При создании экземпляра класса **Vue** ему передаётся объект с настройками. Поле **el** этого объекта как раз отвечает за привязку к html-элементу:

```
...
<body>
  <div id="app"></div>
  ...
</body>
...
```

```
//script.js
const app = new Vue({
  el: '#app'
});
```

Теперь можно добавить данные в объект **app**. Данные передаются в поле **data** объекта настроек:

```
//script.js
const app = new Vue({
  el: '#app',
  data: {
    name: 'Geek'
  }
});
```

К данным можно обратиться из html, обернув имя поля в двойные фигурные скобки. Такая запись называется mustache-синтаксисом:

```
...
<div id="app">
  <h1>Hello, {{ name }}</h1>
```

```
</div>
...
```

Теперь если мы откроем страницу в браузере, то увидим надпись «Hello, Geek».

Запись внутри фигурных скобок – обычный JavaScript, а **name** – переменная. Поэтому любые методы и операции работают и внутри шаблона:

```
...
<div id="app">
  <h1>Hello, {{ name.toUpperCase() }}</h1>
</div>
...
```

В этом случае на странице мы увидим «Hello, GEEK».

Связывание данных

Связывание данных – ключевая особенность Vue.js. Оно позволяет синхронизировать данные и их отображение. Рассмотрим его на примере.

Немного изменим html и добавим поле ввода для указания имени:

```
...
<div id="app">
  <input type="text" />
  <p>Hi, my name is {{ name }}</p>
</div>
...
```

Теперь привяжем значение **name** к полю ввода. Для этого в Vue используется атрибут **v-model**:

```
...
<div id="app">
  <input type="text" v-model="name" />
  <p>Hi, my name is {{ name }}</p>
</div>
...
```

Теперь при изменении значения в поле ввода будет меняться значение **name** и, как следствие, содержимое тэга **<p>**, причём изменение происходит в реальном времени. Эта особенность упрощает работу с данными и помогает избавиться от лишних обработчиков событий.

Условия

Vue позволяет скрывать и показывать элементы в зависимости от условий. Если в нашем примере удалить значение поля **name**, на странице всё равно останется надпись «Hi, my name is». Будем скрывать её, если в **name** записана пустая строка. Для этого к тегу **<p>** добавим атрибут **v-if**:

```
...
<div id="app">
  <input type="text" v-model="name" />
  <p v-if="name.length != 0">Hi, my name is {{ name }}</p>
</div>
...
```

Теперь элемент **<p>** будет отображаться только тогда, когда длина строки, записанной в **name**, не равна 0. Причём элемент не просто скрывается, когда условие **v-if** не выполняется, он полностью отсутствует в DOM.

Списки

Иногда нужно вывести не одно значение, а несколько. Например, есть массив имён, и нам нужно построить маркированный список элементов этого массива. Для генерации повторяемого кода html-шаблона используется атрибут **v-for**.

Рассмотрим это на примере. Удалим все старые данные и разметку. Добавим в данные массив **names**:

```
//script.js
const app = new Vue({
  el: '#app',
  data: {
    names: ['Frodo', 'Sam', 'Meriadoc', 'Peregrin']
  }
});
```

Теперь добавим в html такую конструкцию:

```
...
<ul>
  <li v-for="name in names">{{ name }}</li>
</ul>
...
```

Для каждого элемента массива **names** создаётся свой элемент **<li>**. Внутри него значение элемента массива будет доступно в переменной **name,** которая будет по очереди указывать на элементы массива. Результат будет таким:

```
...
<ul>
  <li>Frodo</li>
  <li>Sam</li>
  <li>Meriadoc</li>
  <li>Peregrin</li>
</ul>
...
```

Повторять можно не только один элемент, но и более сложные вложенные конструкции:

```
...
<ul>
  <li v-for="name in names">
    <pre>{{ name }}</pre>
  </li>
</ul>
...
```

События

Для обработки отслеживания событий на элементе используется атрибут **v-on**. Но прежде чем навешивать (то есть связывать с элементом) событие, нужно позаботиться об обработчике. Добавим поле **methods** в объект настроек. Там будут перечислены функции, доступные из шаблона:

```
//script.js
const app = new Vue({
  el: '#app',
  data: {
    names: ['Frodo', 'Sam', 'Meriadoc', 'Peregrin']
  },
  methods: {}
});
```

Добавим метод обработки клика **clickHandler()**. Пока что будем просто выводить в консоль слово **click**:

```
const app = new Vue({
  el: '#app',
  data: {
    names: ['Frodo', 'Sam', 'Meriadoc', 'Peregrin']
  },
  methods: {
    clickHandler() {
      console.log('click');
    }
  }
});
```

Теперь привяжем вызов этого метода к клику на элемент списка:

```
...
<ul>
  <li v-for="name in names" v-on:click="clickHandler">{{ name }}</li>
</ul>
...
```

Запись можно немного сократить, заменив **v-on** на **@**:

```
...
<ul>
  <li v-for="name in names" @click="clickHandler">{{ name }}</li>
</ul>
...
```

Разумеется, отслеживать можно не только клик, но и другие события, например **@keyup**.

Иногда нужно выполнить какое-то действие сразу, как только загрузится приложение, не дожидаясь действий пользователя. Для этого можно использовать метод **mounted**. Эта функция записывается в одноимённое поле объекта настроек и срабатывает сразу после загрузки приложения.

```
const app = new Vue({
  el: '#app',
  data: {
    names: ['Frodo', 'Sam', 'Meriadoc', 'Peregrin']
  },
  methods: {
    clickHandler() {
      console.log('click');
    }
  },
  mounted() {
    // Сработает сразу
  }
});
```

Вычисляемые свойства

Часто перед тем, как выводить данные, их нужно преобразовать. Это можно сделать внутри конструкции {{ ... }}, но при таком подходе код становится трудночитаемым. Хороший подход – вынести преобразование данных отдельно. Для этого используются вычисляемые свойства.

Вернёмся к знакомому нам примеру:

```
...
<div id="app">
  <input type="text" v-model="name" />
  <p>Hi, my name is {{ name }}</p>
</div>
...
```

```
const app = new Vue({
  el: '#app',
  data: {
    name: 'Geek'
```

```
  }
});
```

Предположим, нам нужно перевести имя в верхний регистр и вставить в **<p>**. Сделаем это через вычисляемые свойства. Для них в объекте настроек есть отдельное поле **computed**:

```
const app = new Vue({
  el: '#app',
  data: {
    name: 'Geek'
  },
  computed: {}
});
```

Вычисляемое свойство – функция, которая обрабатывает исходные данные и возвращает новое значение. Назовём наше свойство **upperCaseName**:

```
const app = new Vue({
  el: '#app',
  data: {
    name: 'Geek'
  },
  computed: {
    upperCaseName() {
      return this.name.toUpperCase();
    }
  }
});
```

Теперь остаётся заменить в шаблоне старое значение на новое:

```
...
<div id="app">
  <input type="text" v-model="name" />
  <p>Hi, my name is {{ upperCaseName }}</p>
</div>
...
```

Практика

Переведём наш интернет-магазин на Vue.js. Прежде всего подключим сам **Vue.js** через CDN и обернём содержимое в ** ** с **id="app"**:

```
<!DOCTYPE html>
<html>
<head>
  <title>eShop</title>
</head>
```

```
<body>
  <div id="app">
    <header>
      <input type="text" class="goods-search" />
      <button class="search-button" type="button">Искать</button>
      <button class="cart-button" type="button">Корзина</button>
    </header>
    <main>
      <div class="goods-list"></div>
    </main>
  </div>

  <script src="https://cdn.jsdelivr.net/npm/vue@2.5.17/dist/vue.js"></script>
  <script src="script.js"></script>
</body>
</html>
```

Теперь в файле **script.js** создадим экземпляр класса **Vue** и привяжем к элементу **#app**:

```
const app = new Vue({
  el: '#app'
});
```

Теперь добавим данные. В нашем случае это будет исходный список товаров, список товаров после фильтрации и содержимое поля поиска:

```
const app = new Vue({
  el: '#app',
  data: {
    goods: [],
    filteredGoods: [],
    searchLine: ''
  }
});
```

Функцию **makeGETRequest** вынесем в отдельный метод:

```
const app = new Vue({
  ...
  methods: {
    makeGETRequest(url, callback) {
      const API_URL =
'https://raw.githubusercontent.com/GeekBrainsTutorial/online-store-api/master/responses';

      var xhr;

      if (window.XMLHttpRequest) {
        xhr = new XMLHttpRequest();
```

```
      } else if (window.ActiveXObject) {
        xhr = new ActiveXObject("Microsoft.XMLHTTP");
      }

      xhr.onreadystatechange = function () {
        if (xhr.readyState === 4) {
          callback(xhr.responseText);
        }
      }

      xhr.open('GET', url, true);
      xhr.send();
    }
  }
});
```

Запрашивать список товаров будем сразу же после загрузки приложения. Воспользуемся методом **mounted()**:

```
const API_URL =
'https://raw.githubusercontent.com/GeekBrainsTutorial/online-store-api/master/responses';

const app = new Vue({
  ...
  mounted() {
    this.makeGETRequest(`${API_URL}/catalogData.json`, (goods) => {
      this.goods = goods;
      this.filteredGoods = goods;
    });
  }
});
```

Теперь добавим в шаблон отрисовку списка товаров из поля **filteredGoods** с помощью **v-for**:

```
...
<main>
  <div class="goods-list">
    <div class="goods-item" v-for="good in filteredGoods">
      <h3>{{ good.product_name }}</h3>
      <p>{{ good.price }}</p>
    </div>
  </div>
</main>
...
```

Практическое задание

Все пункты выполняются с использованием Vue.js.

1. Добавить методы и обработчики событий для поля поиска. Создать в объекте данных поле **searchLine** и привязать к нему содержимое поля ввода. На кнопку **Искать** добавить обработчик клика, вызывающий метод **FilterGoods**.
2. Добавить корзину. В html-шаблон добавить разметку корзины. Добавить в объект данных поле **isVisibleCart**, управляющее видимостью корзины.
3. * Добавлять в **.goods-list** заглушку с текстом «Нет данных» в случае, если список товаров пуст.

Дополнительные материалы

1. Видеокурс по основам Vue.js.

Используемая литература

1. Официальная документация Vue.js.