

Раздел

[Веб-компоненты](#)

Навигация по уроку

Пример: «time-formatted»

Наблюдение за атрибутами

Порядок рендеринга

Модифицированные
встроенные элементы


Ссылки

Итого

Задачи (1)

Комментарии

Поделиться

[Редактировать на GitHub](#) → [Веб-компоненты](#) 13-го января 2020

Пользовательские элементы (Custom Elements)

Мы можем создавать пользовательские HTML-элементы, описываемые нашим классом, со своими методами и свойствами, событиями и так далее.

Как только пользовательский элемент определён, мы можем использовать его наравне со встроенными HTML-элементами.

Это замечательно, ведь словарь HTML-тегов богат, но не бесконечен. Не существует `<easy-tabs>`, `<sliding-carousel>`, `<beautiful-upload>` ... Просто подумайте о любом другом теге, который мог бы нам понадобиться.

Мы можем определить их с помощью специального класса, а затем использовать, как если бы они всегда были частью HTML.

Существует два вида пользовательских элементов:

1. **Автономные пользовательские элементы** – «полностью новые» элементы, расширяющие абстрактный класс `HTMLElement`.
2. **Пользовательские встроенные элементы** – элементы, расширяющие встроенные, например кнопку `HTMLButtonElement` и т.п.

Сначала мы разберёмся с автономными элементами, а затем перейдём к пользовательским встроенным.

Чтобы создать пользовательский элемент, нам нужно сообщить браузеру ряд деталей о нём: как его показать, что делать, когда элемент добавляется или удаляется со страницы и т.д.

Это делается путём создания класса со специальными методами. Это просто, так как существует всего несколько методов, и все они являются необязательными.

Вот набросок с полным списком:

```
1 class MyElement extends HTMLElement {
2   constructor() {
3     super();
4     // элемент создан
5   }
6
7   connectedCallback() {
8     // браузер вызывает этот метод при добавлении элеме
9     // (может вызываться много раз, если элемент многок
10  }
11
12   disconnectedCallback() {
13     // браузер вызывает этот метод при удалении элемент
14     // (может вызываться много раз, если элемент многок
15   }
16
17   static get observedAttributes() {
18     return /* массив имён атрибутов для отслеживания и
19   }
20
21   attributeChangedCallback(name, oldValue, newValue) {
22     // вызывается при изменении одного из перечисленных
23   }
24
25   adoptedCallback() {
26     // вызывается, когда элемент перемещается в новый д
27     // (происходит в document.adoptNode, используется о
28   }
29 }
```

```
30 // у элемента могут быть ещё другие методы и свойства
31 }
```

Раздел

Веб-компоненты

Навигация по уроку

Пример: «time-formatted»

Наблюдение за атрибутами

Порядок рендеринга

Модифицированные
встроенные элементы

Ссылки

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



После этого нам нужно зарегистрировать элемент:



```
1 // сообщим браузеру, что <my-element> обслуживается наш
2 customElements.define("my-element", MyElement);
```

Теперь для любых HTML-элементов с тегом `<my-element>` создаётся экземпляр `MyElement` и вызываются вышеупомянутые методы. Также мы можем использовать `document.createElement('my-element')` в JavaScript.

Имя пользовательского элемента должно содержать дефис -

Имя пользовательского элемента должно содержать дефис -, например, `my-element` и `super-button` – валидные имена, а `myelement` – нет.

Это чтобы гарантировать отсутствие конфликтов имён между встроенными и пользовательскими элементами HTML.

Пример: «time-formatted»

Например, элемент `<time>` уже существует в HTML для даты/времени. Но сам по себе он не выполняет никакого форматирования. Давайте создадим элемент `<time-formatted>`, который отображает время в удобном формате с учётом языка:

```
1 <script>
2 class TimeFormatted extends HTMLElement { // (1)
3
4   connectedCallback() {
5     let date = new Date(this.getAttribute('datetime') |
6
7     this.innerHTML = new Intl.DateTimeFormat("default",
8       year: this.getAttribute('year') || undefined,
9       month: this.getAttribute('month') || undefined,
10      day: this.getAttribute('day') || undefined,
11      hour: this.getAttribute('hour') || undefined,
12      minute: this.getAttribute('minute') || undefined,
13      second: this.getAttribute('second') || undefined,
14      timeZoneName: this.getAttribute('time-zone-name')
15    }).format(date);
16   }
17
18 }
19
20 customElements.define("time-formatted", TimeFormatted);
21 </script>
22
23 <!-- (3) -->
24 <time-formatted datetime="2019-12-01"
25   year="numeric" month="long" day="numeric"
26   hour="numeric" minute="numeric" second="numeric"
27   time-zone-name="short"
28 ></time-formatted>
```

1 декабря 2019 г., 3:00:00 GMT+3

1. Класс имеет только один метод `connectedCallback()` – браузер вызывает его, когда элемент `<time-formatted>` добавляется на страницу (или когда HTML-парсер обнаруживает его), и он использует встроенный форматировщик данных `Intl.DateTimeFormat`, хорошо

Раздел

Веб-компоненты

Навигация по уроку

Пример: «time-formatted»

Наблюдение за атрибутами

Порядок рендеринга

Модифицированные
встроенные элементы

Ссылки

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



поддерживаемый в браузерах, чтобы показать красиво отформатированное время.

2. Нам нужно зарегистрировать наш новый элемент, используя `customElements.define(tag, class)`.
3. И тогда мы сможем использовать его везде.

i Обновление пользовательских элементов

Если браузер сталкивается с элементами `<time-formatted>` до `customElements.define`, то это не ошибка. Но элемент пока неизвестен, как и любой нестандартный тег.

Такие «неопределённые» элементы могут быть стилизованы с помощью CSS селектора `:not(:defined)`.

Когда вызывается `customElements.define`, они «обновляются»: для каждого создаётся новый экземпляр `TimeFormatted` и вызывается `connectedCallback`. Они становятся `:defined`.

Чтобы получить информацию о пользовательских элементах, есть следующие методы:

- `customElements.get(name)` – возвращает класс пользовательского элемента с указанным именем `name`,
- `customElements.whenDefined(name)` – возвращает промис, который переходит в состояние «успешно выполнен» (без значения), когда определён пользовательский элемент с указанным именем `name`.

i Рендеринг происходит в `connectedCallback`, не в `constructor`

В приведённом выше примере содержимое элемента рендерится (создаётся) в `connectedCallback`.

Почему не в `constructor`?

Причина проста: когда вызывается `constructor`, делать это слишком рано. Экземпляр элемента создан, но на этом этапе браузер ещё не обработал/назначил атрибуты: вызовы `getAttribute` вернули бы `null`. Так что мы не можем рендерить здесь.

Кроме того, если подумать, это лучше с точки зрения производительности – отложить работу до тех пор, пока она действительно не понадобится.

`connectedCallback` срабатывает, когда элемент добавляется в документ. Не просто добавляется к другому элементу как дочерний, но фактически становится частью страницы. Таким образом, мы можем построить отдельный DOM, создать элементы и подготовить их для последующего использования. Они будут рендериться только тогда, когда попадут на страницу.

Наблюдение за атрибутами

В текущей реализации `<time-formatted>` после того, как элемент отрендерился, дальнейшие изменения атрибутов не дают никакого эффекта. Это странно для HTML-элемента. Обычно, когда мы изменяем атрибут, например `a.href`, мы ожидаем, что изменение будет видно сразу. Так что давайте исправим это.

Мы можем наблюдать за атрибутами, поместив их список в статический геттер `observedAttributes()`. При изменении таких атрибутов вызывается `attributeChangedCallback`. Он срабатывает не для любого атрибута по соображениям производительности.

Вот новый `<time-formatted>`, который автоматически обновляется при изменении атрибутов:

1 `<script>`



Раздел

Веб-компоненты

Навигация по уроку

Пример: «time-formatted»

Наблюдение за атрибутами

Порядок рендеринга

Модифицированные
встроенные элементы

Ссылки

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



```
2 class TimeFormatted extends HTMLElement {
3
4   render() { // (1)
5     let date = new Date(this.getAttribute('datetime') |
6
7     this.innerHTML = new Intl.DateTimeFormat("default",
8       year: this.getAttribute('year') || undefined,
9       month: this.getAttribute('month') || undefined,
10      day: this.getAttribute('day') || undefined,
11      hour: this.getAttribute('hour') || undefined,
12      minute: this.getAttribute('minute') || undefined,
13      second: this.getAttribute('second') || undefined,
14      timeZoneName: this.getAttribute('time-zone-name')
15    }).format(date);
16  }
17
18  connectedCallback() { // (2)
19    if (!this.rendered) {
20      this.render();
21      this.rendered = true;
22    }
23  }
24
25  static get observedAttributes() { // (3)
26    return ['datetime', 'year', 'month', 'day', 'hour',
27  }
28
29  attributeChangedCallback(name, oldValue, newValue) {
30    this.render();
31  }
32
33 }
34
35 customElements.define("time-formatted", TimeFormatted);
36 </script>
37
38 <time-formatted id="elem" hour="numeric" minute="numeri
39
40 <script>
41 setInterval(() => elem.setAttribute('datetime', new Dat
42 </script>
```

10:37:03

1. Логика рендеринга перенесена во вспомогательный метод `render()`.
2. Мы вызываем его один раз, когда элемент вставляется на страницу.
3. При изменении атрибута, указанного в `observedAttributes()`, вызывается `attributeChangedCallback`.
4. ...и происходит ререндеринг элемента.
5. В конце мы легко создаём живой таймер.

Порядок рендеринга

Когда HTML-парсер строит DOM, элементы обрабатываются друг за другом, родители до детей. Например, если у нас есть `<outer><inner></inner></outer>`, то элемент `<outer>` создаётся и включается в DOM первым, а затем `<inner>`.

Это приводит к важным последствиям для пользовательских элементов.

Например, если пользовательский элемент пытается получить доступ к `innerHTML` в `connectedCallback`, он ничего не получает:

```
1 <script>
2 customElements.define('user-info', class extends HTMLEl
3
4   connectedCallback() {
5     alert(this.innerHTML); // пусто (*)
6   }
```

Раздел

Веб-компоненты

Навигация по уроку

Пример: «time-formatted»

Наблюдение за атрибутами

Порядок рендеринга

Модифицированные
встроенные элементы

Ссылки

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



```
7
8 });
9 </script>
10
11 <user-info>Джон</user-info>
```

Если вы запустите это, `alert` будет пуст.

Это происходит именно потому, что на этой стадии ещё не существуют дочерние элементы, DOM не завершён. HTML-парсер подключил пользовательский элемент `<user-info>` и теперь собирается перейти к его дочерним элементам, но пока не сделал этого.

Если мы хотим передать информацию в пользовательский элемент, мы можем использовать атрибуты. Они доступны сразу.

Или, если нам действительно нужны дочерние элементы, мы можем отложить доступ к ним, используя `setTimeout` с нулевой задержкой.

Это работает:

```
1 <script>
2   customElements.define('user-info', class extends HTMLLEl
3
4     connectedCallback() {
5       setTimeout(() => alert(this.innerHTML)); // Джон (*)
6     }
7
8   });
9 </script>
10
11 <user-info>Джон</user-info>
```

Теперь `alert` в строке (*) показывает «Джон», поскольку мы запускаем его асинхронно, после завершения парсинга HTML. Мы можем обработать дочерние элементы при необходимости и завершить инициализацию.

С другой стороны, это решение также не идеально. Если вложенные пользовательские элементы тоже используют `setTimeout` для инициализации, то они встанут в очередь: первым запускается внешний `setTimeout`, а затем внутренний.

Так что внешний элемент завершает инициализацию раньше внутреннего.

Продемонстрируем это на примере:

```
1 <script>
2   customElements.define('user-info', class extends HTMLLEl
3     connectedCallback() {
4       alert(`${this.id} connected.`);
5       setTimeout(() => alert(`${this.id} initialized.`));
6     }
7   });
8 </script>
9
10 <user-info id="outer">
11   <user-info id="inner"></user-info>
12 </user-info>
```

Порядок вывода:

1. outer connected.
2. inner connected.
3. outer initialized.
4. inner initialized.

Мы ясно видим, что внешний элемент `outer` завершает инициализацию (3) до внутреннего `inner` (4).

Раздел

Веб-компоненты

Навигация по уроку

Пример: «time-formatted»

Наблюдение за атрибутами

Порядок рендеринга

Модифицированные
встроенные элементы

Ссылки

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



Нет встроенного колбэка, который срабатывает после того, как вложенные элементы готовы. Если нужно, мы можем реализовать подобное самостоятельно. Например, внутренние элементы могут отправлять события наподобие `initialized`, а внешние могут слушать и реагировать на них.

Модифицированные встроенные элементы

Новые элементы, которые мы создаём, такие как `<time-formatted>`, не имеют связанной с ними семантики. Они не известны поисковым системам, а устройства для людей с ограниченными возможностями не могут справиться с ними.

Но такие вещи могут быть важны. Например, поисковой системе было бы интересно узнать, что мы показываем именно время. А если мы делаем специальный вид кнопки, почему не использовать существующую функциональность `<button>`?

Мы можем расширять и модифицировать встроенные HTML-элементы, наследуя их классы.

Например, кнопки `<button>` являются экземплярами класса `HTMLButtonElement`, давайте построим элемент на его основе.

1. Унаследуем `HTMLButtonElement` нашим классом:

```
1 class HelloButton extends HTMLButtonElement { /* мето
```

2. Предоставим третий аргумент в `customElements.define`, указывающий тег:

```
1 customElements.define('hello-button', HelloButton, {e
```

Бывает, что разные теги имеют одинаковый DOM-класс, поэтому указание тега необходимо.

3. В конце, чтобы использовать наш пользовательский элемент, вставим обычный тег `<button>`, но добавим к нему `is="hello-button"`:

```
1 <button is="hello-button">...</button>
```

Вот полный пример:

```
1 <script>
2 // Кнопка, говорящая "привет" по клику
3 class HelloButton extends HTMLButtonElement {
4   constructor() {
5     super();
6     this.addEventListener('click', () => alert("Привет!
7   }
8 }
9
10 customElements.define('hello-button', HelloButton, {ext
11 </script>
12
13 <button is="hello-button">Нажми на меня</button>
14
15 <button is="hello-button" disabled>Отключена</button>
```

Нажми на меня Отключена

Наша новая кнопка расширяет встроенную. Так что она сохраняет те же стили и стандартные возможности, наподобие атрибута `disabled`.

Ссылки

Раздел

Веб-компоненты

Навигация по уроку

Пример: «time-formatted»

Наблюдение за атрибутами

Порядок рендеринга

Модифицированные
встроенные элементы

Ссылки

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



Итого

Есть два типа пользовательских элементов:

1. «Автономные» – новые теги, расширяющие `HTMLElement`.

Схема определения:

```
1 class MyElement extends HTMLElement {
2   constructor() { super(); /* ... */ }
3   connectedCallback() { /* ... */ }
4   disconnectedCallback() { /* ... */ }
5   static get observedAttributes() { return /* ... */ }
6   attributeChangedCallback(name, oldValue, newValue)
7   adoptedCallback() { /* ... */ }
8 }
9 customElements.define('my-element', MyElement);
10 /* <my-element> */
```

2. «Модифицированные встроенные элементы» – расширения существующих элементов.

Требуют ещё один аргумент в `.define` и атрибут `is="..."` в HTML:

```
1 class MyButton extends HTMLButtonElement { /*...*/ }
2 customElements.define('my-button', MyElement, {extend
3 /* <button is="my-button"> */
```

Пользовательские элементы широко поддерживаются браузерами. Edge немного отстаёт, но есть полифил <https://github.com/webcomponents/webcomponentsjs>.

✓ Задачи

Элемент "живой таймер"

У нас уже есть элемент `<time-formatted>`, показывающий красиво отформатированное время.

Создайте элемент `<live-timer>`, показывающий текущее время:

1. Внутри он должен использовать `<time-formatted>`, не дублировать его функциональность.
2. Должен тикать (обновляться) каждую секунду.
3. На каждом тике должно генерироваться пользовательское событие с именем `tick`, содержащее текущую дату в `event.detail` (смотрите главу [Генерация пользовательских событий](#)).

Использование:

```
1 <live-timer id="elem"></live-timer>
2
3 <script>
4   elem.addEventListener('tick', event => console.log(ev
5 </script>
```

Демо:

10:37:02

[Открыть песочницу для задачи.](#)

решение

Раздел

[Веб-компоненты](#)

Навигация по уроку

Пример: «time-formatted»

Наблюдение за атрибутами

Порядок рендеринга

Модифицированные
встроенные элементы

Ссылки

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Комментарии

перед тем как писать...

© 2007–2020 Илья Кантор | [о проекте](#) | [связаться с нами](#) | [пользовательское соглашение](#) | [политика конфи](#)

