

ОБРАБОТКА СОБЫТИЙ



АЛЕКСЕЙ ДАЦКОВ



АЛЕКСЕЙ ДАЦКОВ

Руководитель разработки проекта в Берито



alex.smap@gmail.com



[alex.smap](https://t.me/alex.smap)



[alex.smap](https://vk.com/alex.smap)



ПЛАН ЗАНЯТИЯ

1. [Events](#)
2. [EventListeners](#)
3. [Prevent Default](#)
4. [Capturing & Bubbling](#)

ЗАДАЧА

Мы делаем CRM систему, а в частности страницу со списком клиентов. И перед нами встала задача организации фильтрации клиентов, представленных на странице:

The image shows a UI mockup for a CRM system. At the top, there is a filter section with the label 'Фильтр' followed by a text input field and an 'Ok' button. Below this, there is a list of three client entries. Each entry consists of a square placeholder icon with an 'X', the client's name and phone number, and a 'Позвонить' (Call) button.

Иконка	Имя	Телефон	Действие
	Иванов Александр	+7960 077 00 00	Позвонить
	Петров Сергей	+7903 077 00 00	Позвонить
	Сидоров Антон	+7928 077 00 00	Позвонить

Поскольку вы уже умеете манипулировать DOM, нам остаётся лишь научиться реагировать на ввод пользователя и на нажатие кнопки «Ok» (плюс не забыть, что часть пользователей используют «Enter» вместо нажатия кнопки «Ok»).

БАЗОВАЯ РАЗМЕТКА

```
1 <div data-widget="filter-widget">
2   <form>
3     <label>Фильтр <input data-id="filter-text" name="filter-text"></label>
4     <button data-action="filter">Ok</button>
5   </form>
6 </div>
7
8 <ul data-section="contact-list">
9   <li data-contact data-contact-id="...">
10     <div data-section="main">
11       
12       <span>Иванов Александр</span><span>+7960 077 00 00</span>
13       <a href="tel:+79600770000" data-phone="79600770000" data-action="call">Звонок</a>
14     </div>
15     <div data-section="details" class="contact-details hidden">
16       Подробная информация о клиенте
17     </div>
18   </li>
19   <li>...</li>
20   <li>...</li>
21 </ul>
```

ДАННЫЕ

```
1  [  
2    {  
3      "id": 1,  
4      "name": "Иванов Александр",  
5      "avatar": "https://placeimg.com/50/50/people",  
6      "phone": "79600770000",  
7      "details": "Подбробнее информация о клиенте"  
8    },  
9    ...  
10 ]
```

ГЕНЕРАЦИЯ DOM

```
1  const data = JSON.parse('-> json here <-');
2  const contactListEl = document.querySelector('[data-section=contact-list]');
3  const contactTag = 'li';
4
5  function buildContactList(contacts, containerEl, elementTag) {
6    containerEl.innerHTML = '';
7    contacts.forEach((o) => {
8      const el = document.createElement(elementTag);
9      el.dataset.contact = '';
10     el.dataset.contactId = o.id;
11     el.innerHTML = `
12     <div data-section="main">
13       
14       <span class="contact-name">${o.name}</span><span class="contact-phone">${o.phone}</span>
15       <a href="tel:>${o.phone}" data-phone="${o.phone}" data-action="call">Звонок</a>
16     </div>
17     <div data-section="details" class="hidden">
18       ${o.details}
19     </div>
20     `;
21     contactListEl.appendChild(el);
22   });
23 }
24 buildContactList(data, contactListEl, contactTag);
```

ФИЛЬТР

Модуль фильтрации будет выглядеть следующим образом:

```
1 export function filterBy(contacts, filterCallback) {  
2   return contacts.filter(filterCallback);  
3 }  
4  
5 export function containsPhone(data, search) {  
6   const clean = search.replace(/[+ ()]/g, ''); // Удаляем +, ' ' и т.д.  
7   return data.startsWith(clean);  
8 }  
9  
10 export function containsText(data, search) {  
11   const clean = search.trim().toLowerCase();  
12   return data.toLowerCase().includes(clean);  
13 }
```

Обратите внимание, данные функции ничего не знают о DOM и Browser Environment и их легко покрыть обычными unit-тестами без всяких зависимостей.



EVENTS

СОБЫТИЯ

События — это специальные объекты, соответствующие интерфейсу `Event`, направляемые другим объектам, соответствующим интерфейсу `EventTarget` (в рамках этой лекции мы их будем называть eventTarget'ами).

Причиной возникновения событий могут быть действия пользователя, сетевое взаимодействие и т.д.

Отправкой событий (dispatching) занимается в основном, браузер, но мы также можем создавать собственные события и программно отправлять их eventTarget'ам.

EVENTTARGET

Интерфейс `EventTarget` выглядит следующим образом:

```
1 interface EventTarget {  
2     void addEventListener(  
3         DOMString type, EventListener? callback,  
4         optional (AddEventListenerOptions or boolean) options  
5     );  
6     void removeEventListener(  
7         DOMString type,  
8         EventListener? callback,  
9         optional (EventListenerOptions or boolean) options  
10    );  
11    boolean dispatchEvent(Event event);  
12 };
```

Т.е. он содержит описание всего трёх методов, которые должны быть реализованы в объекте.

EVENTTARGET

- `addEventListener` – позволяет добавить «слушателя» события
- `removeEventListener` – позволяет удалить «слушателя» события
- `dispatchEvent` – позволяет отправить событие eventTarget'у

Как вы видите, среди методов нет метода, позволяющего получить список «слушателей» события.

EVENTTARGET И NODE

С прошлой лекции мы с вами видели цепочку наследования интерфейсов `HTMLElement` -> `Element` -> `Node` -> `EventTarget`. Что означает, что любая `Node` является `EventTarget` 'ом, но не любой `EventTarget` является `Node`.

Т.е. нам достаточно узнать тип события и добавить слушателя этого события (объект, соответствующий интерфейсу `EventListener`) на любой элемент.

EVENT TYPES

Описание типов событий можно найти на MDN:

<https://developer.mozilla.org/en-US/docs/Web/Events>.

Поскольку нас интересует клик по кнопке «Ok», то нам подойдёт событие `click`.

Попробуем добавить обработчик:

```
1 | const filterWidgetEl = document.querySelector('[data-widget=filter-widget]');  
2 | const filterBtnEl = filterWidgetEl.querySelector('[data-action=filter]');  
3 | filterBtnEl.addEventListener('click', <eventListener>);
```

Важно: внимательно изучите страницу с описанием типов событий.



EVENTLISTENERS

EVENTLISTENER

В качестве `eventListener`'а должен быть указан объект, удовлетворяющий интерфейсу `EventListener`:

```
1  callback interface EventListener {  
2    void handleEvent(Event event);  
3  };
```

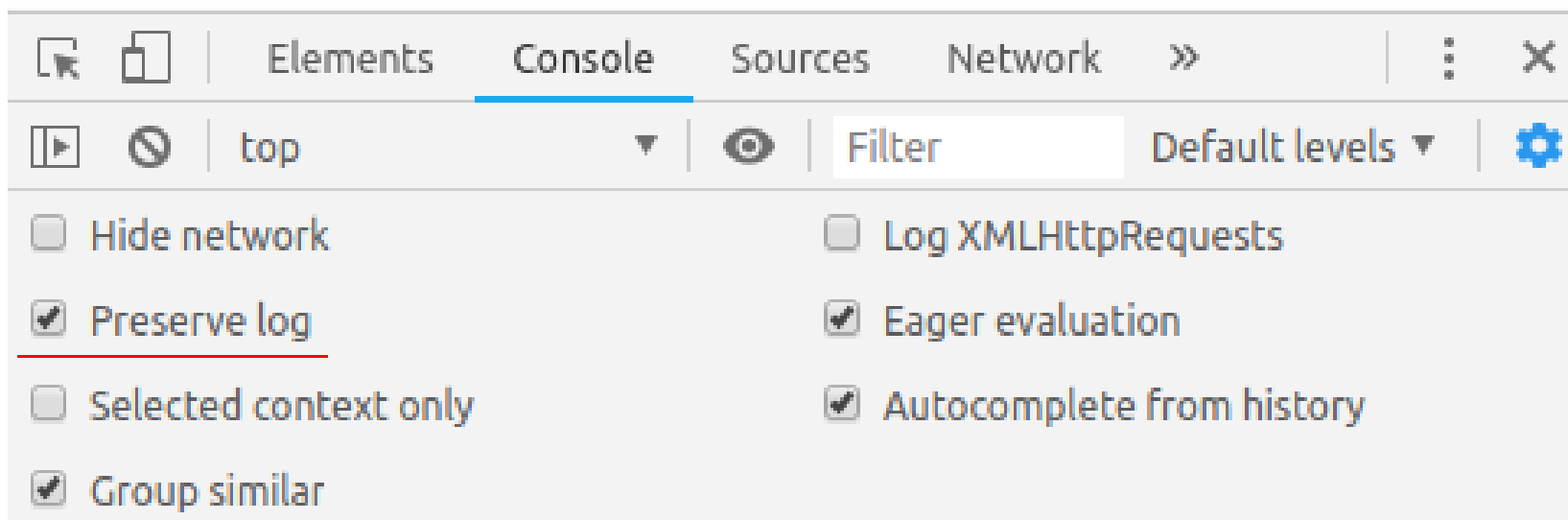
Т.е. нужно передать объект, в котором есть метод `handleEvent` (но поскольку в этом интерфейсе всего один метод, нам достаточно передать просто функцию):

```
1  const filterWidgetEl = document.querySelector('[data-widget=filter-widget]');  
2  const filterBtnEl = filterWidgetEl.querySelector('[data-action=filter]');  
3  const filterTextEl = filterWidgetEl.querySelector('[data-id=filter-text]');  
4  filterBtnEl.addEventListener('click', (event) => {  
5    console.log(event);  
6    console.log(filterTextEl);  
7  });
```


EVENTLISTENER

Страница перезагружается, и поэтому сообщения `console.log` не сохраняются.

Поставим галочку `Preserve Log` для сохранения сообщений:

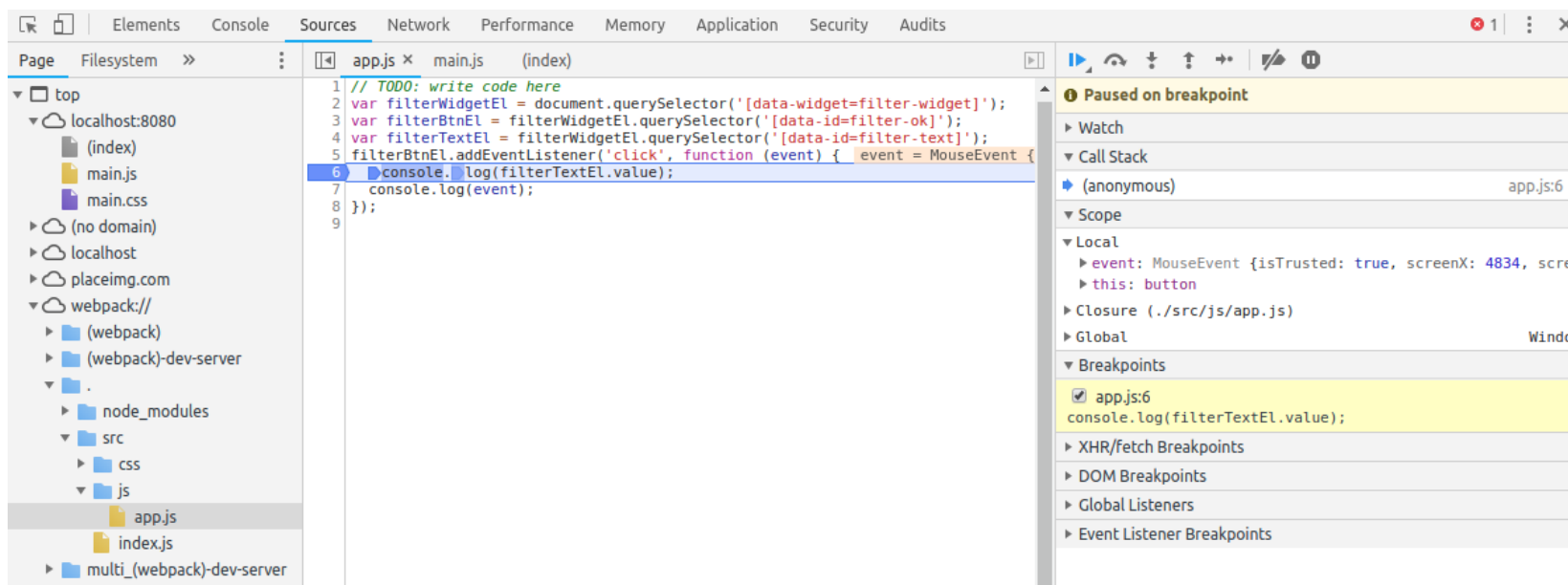


DEBUGGER

`console.log`, конечно, хорошо, но лучше научиться пользоваться дебаггером, который позволит вам анализировать ситуацию в конкретной точке вашей программы + иметь возможность влиять на происходящее (без необходимости перезапуска всего действия).

Поскольку Webpack «собирает» всё в бандл, мы не увидим наш скрипт в «обычном местоположении» — `localhost:8080`, он будет «спрятан» в секции `webpack://` -> `.` -> `src` -> `js`:

DEBUGGER



Организуется данная функциональность благодаря SourceMap'ингу, когда исходным файлам сопоставляются секции в скомпилированном.

Для этого в `webpack.config.js` нужно прописать:

```
devtool: 'source-map',
```

MOUSEEVENT

Как видно в логах или debugger'е, объект `event` соответствует интерфейсу `MouseEvent`, который в свою очередь, наследуется от `Event`.

EVENT LISTENER BREAKPOINTS

Не обязательно ставить breakpoint вручную, можно его выставить с помощью панельки **Event Listener Breakpoints**:

▼ Event Listener Breakpoints

- ▶ ☐ Animation
- ▶ ☐ Canvas
- ▶ ☐ Clipboard
- ▶ ☐ Control
- ▶ ☐ DOM Mutation
- ▶ ☐ Device
- ▶ ☐ Drag / drop
- ▶ ☐ Geolocation
- ▶ ☐ Keyboard
- ▶ ☐ Load
- ▶ ☐ Media
- ▼ ☒ Mouse
 - ☐ auxclick
 - ☒ click
 - ☐ dblclick
 - ☐ mousedown
 - ☐ mouseup
 - ☐ mouseover
 - ☐ mousemove
 - ☐ mouseout
 - ☐ mouseenter
 - ☐ mouseleave
 - ☐ mousewheel
 - ☐ wheel
 - ☐ contextmenu



ПЕРЕЗАГРУЗКА СТРАНИЦЫ

Вернёмся к нашей задаче: мы действительно получаем уведомление браузера о произошедшем событии, но при этом страница перезагружается.

Вопрос к аудитории: что означает для нас перезагрузка страницы?



ПЕРЕЗАГРУЗКА СТРАНИЦЫ

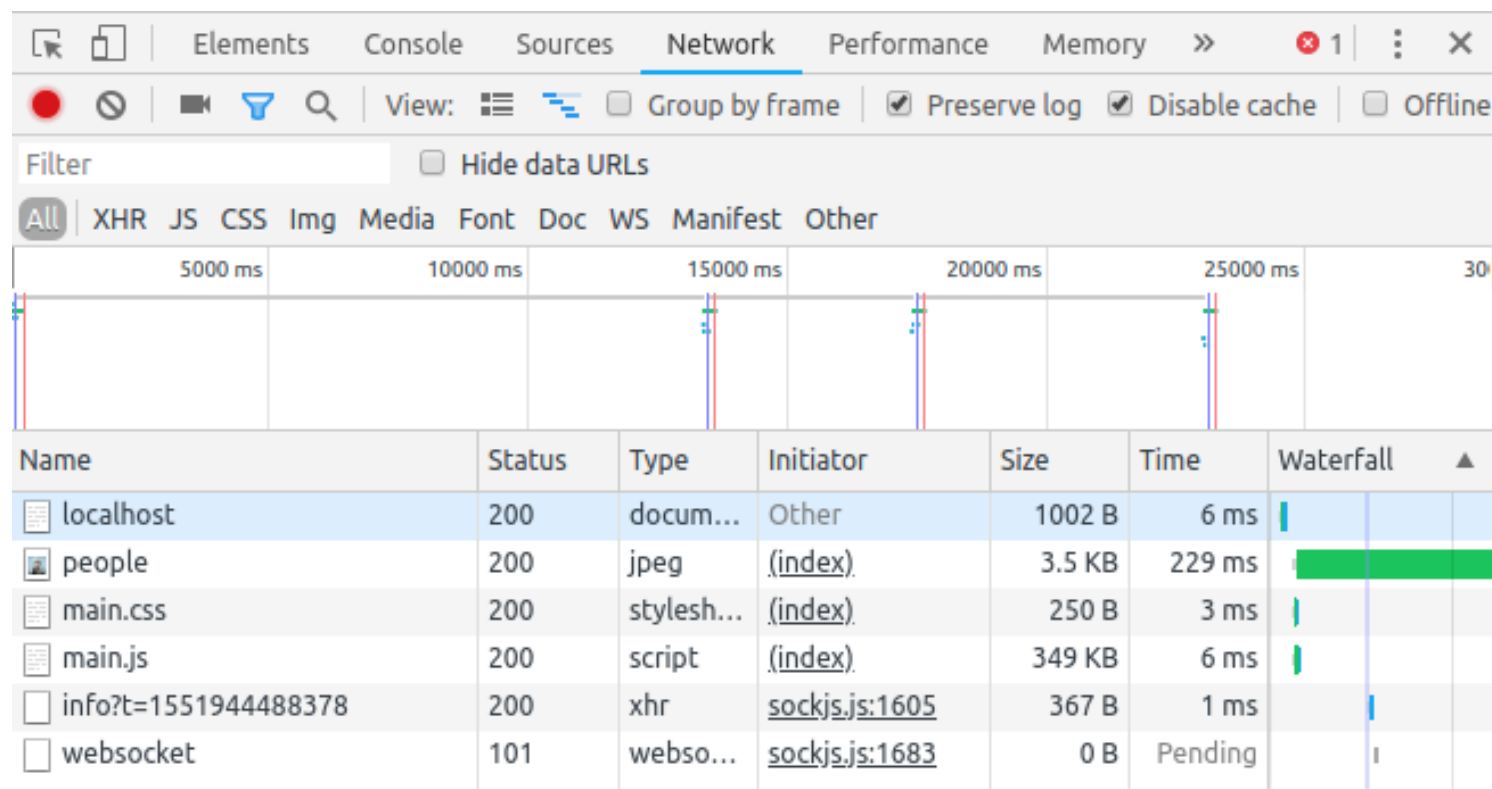
Перезагрузка страницы (если рассматривать упрощённо) приводит в первую очередь к тому, что DOM и наши JS-скрипты выгружаются из памяти и весь процесс построения страницы начинается заново.

Это совсем не то, чего бы нам хотелось, потому что мы теряем контроль над страницей (в том числе данные, введённые в поля, очищаются).

ПЕРЕЗАГРУЗКА СТРАНИЦЫ

Перейдём на вкладку **Network** (не забудьте отключить breakpoint).





При первой загрузке она выглядит вот так:



Примечание: не забудьте поставить галочки **Preserve Log** и здесь.

ОТПРАВКА ФОРМЫ








Если внимательно присмотреться, то можно увидеть, что на самом деле происходит не перезагрузка страницы, а отправка формы, которая и приводит к перезагрузке страницы:

Name	× Headers Preview Response Cookies Timing
 ?filter-text=%D0%98%D0%B2%D0%...	▶ General
 people	▶ Response Headers (7)
 main.css	▶ Request Headers (11)
 main.js	▼ Query String Parameters view source view URL encoded
<input type="checkbox"/> info?t=1551944726026	filter-text: ИВАНОВ
<input type="checkbox"/> websocket	

ОТПРАВКА ФОРМЫ

Происходит это в соответствии со спецификацией: если внутри формы есть кнопка (в нашем случае без атрибута `type`), то нажатие этой кнопки приводит к отправке формы.

Куда и как отправляется форма, определяется атрибутами `action` и `method` самой формы. Если они не указаны, то отправка происходит на тот же адрес, с которого страница загружена:

Name	× Headers Preview Response Cookies Timing
 ?filter-text=%D0%98%D0%B2%D0%...	▼ General
 people	Request URL: http://localhost:8080/?filter-text=%D0%98%D0%B2%D0%B0%D0%BD%D0%BE%D0%B2
 main.css	Request Method: GET
 main.js	Status Code:  200 OK
 info?t=1551944726026	Remote Address: 127.0.0.1:8080
 websocket	Referrer Policy: no-referrer-when-downgrade

ОТПРАВКА ФОРМЫ

Стоп-стоп, но с чего вдруг кнопка превращается в кнопку для отправки формы, где это написано?

[Спецификация, раздел, посвящённый формам:](#)

Keyword	State	Brief description
<code>submit</code>	<code>submit button</code>	Submits the form.
<code>reset</code>	<code>reset button</code>	Resets the form.
<code>button</code>	<code>Button</code>	Does nothing.

The missing value default is the `submit button` state.

Т.е. по факту, если элемент `button` располагается внутри формы и у него не указан атрибут `type`, то автоматически атрибут `type` становится равным `submit`.



СПЕЦИФИКАЦИЯ

Мы настоятельно рекомендуем вам периодически заглядывать в спецификацию:

- это сэкономит вам многие часы поиска
- прокачает вас как специалиста

Но есть нюанс — в браузерах встречаются ошибки реализации спецификации (с этим уже ничего не поделаешь).

DEFAULT ACTIONS

Итак, несмотря на то, что мы перехватываем событие, форма всё равно отправляется. Это называется **Default Actions** и ряд событий содержат предопределённое поведение по умолчанию:

- клик на кнопке отправки в форме
- клик на ссылке
- нажатие клавиш клавиатуры в поле ввода

Interface

Event

Targets

Element

Default Action

Varies (send the content of the form to the server)



PREVENT DEFAULT

PREVENT DEFAULT

Для того, чтобы попросить браузер отменить поведение по умолчанию, мы можем вызвать метод `preventDefault()` на объекте события:

```
1 filterBtnEl.addEventListener('click', (event) => {  
2     event.preventDefault();  
3     console.log(event);  
4 });
```

После этих действий форма действительно перестанет отправляться.

MULTIPLE LISTENERS

Давайте попробуем на одно событие навесить несколько обработчиков и посмотреть:

1. В каком порядке они будут выполняться
2. Не будет ли мешать `preventDefault`

```
1 // ниже первого обработчика
2 filterBtnEl.addEventListener('click', (event) => {
3   console.log('logger listener');
4   // TODO: send data to Google Analytics & Yandex Metrica
5 })
```


MULTIPLE LISTENERS

1. Порядок вызова в данном случае всегда чётко определён, соответствует он порядку добавления `eventListener`’ов.
2. `preventDefault` отменяет только Default Action и не мешает другим `eventListener`’ам.

Информация о том, отменил ли кто-то из обработчиков действие по умолчанию, содержится в поле `defaultPrevented`:

```
1 // ниже первого обработчика
2 filterBtnEl.addEventListener('click', (event) => {
3     console.log('logger listener');
4     console.log(event.defaultPrevented);
5     // TODO: send data to Google Analytics & Yandex Metrica
6 })
```

DEFAULTPREVENTED

В спецификациях многие свойства помечены как `readonly`:

```
readonly attribute boolean defaultPrevented;
```

Это значит, что вы не можете сделать `unpreventDefault`:

```
1 filterBtnEl.addEventListener('click', (event) => {  
2   console.log('logger listener');  
3   console.log(event.defaultPrevented);  
4   event.defaultPrevented = false;  
5   console.log(event.defaultPrevented); // всё равно true  
6 });
```

DEFAULTPREVENTED

Важные замечания:

1. В отличие от `Object.freeze`, попытка назначить значение `readonly` атрибуту никакой ошибки не вызовет (в не-strict режиме)
2. Повторные вызовы `preventDefault` в других (либо этом же обработчике) также ни к каким ошибкам не приведут, поэтому вы очень редко увидите код, который проверяет свойство `defaultPrevented`

ОТСТУПЛЕНИЕ: REMOVE EVENTLISTENER

Удаление `eventListener`'ов достаточно редкая операция, но содержит ряд тонкостей. Ключевой из которых является то, что при удалении вы обязаны указывать ту же самую функцию, что указывали при добавлении `eventListener`'а:

```
1 filterBtnEl.addEventListener('click', (event) => {  
2   console.log('logger listener');  
3 });  
4 filterBtnEl.removeEventListener('click', (event) => {  
5   console.log('logger removed');  
6 });
```

Код выше работать не будет. Потому что указаны две разные функции (это два разных объекта).

REMOVE EVENTLISTENER

```
1  const listener = (event) => {  
2    // TODO:  
3  };  
4  
5  filterBtnEl.addEventListener('click', listener);  
6  filterBtnEl.removeEventListener('click', listener);
```

ЧТЕНИЕ ТЕКСТА

Итак, мы научились перехватывать событие, отменять действие по умолчанию. Осталось только прочесть значение поля ввода и написать логику фильтрации элементов (а позже перестроить DOM).

В принципе, всё просто: читаем свойство `value` у поля ввода:

```
1 filterBtnEl.addEventListener('click', (event) => {  
2   event.preventDefault();  
3   const text = filterTextEl.value;  
4   // TODO: логика обработки значения и фильтрации  
5 });
```

РЕАГИРУЕМ НА ENTER

Наверное, для этого надо научиться реагировать на события клавиатуры. Такие действительно есть — `keydown` и т.д.

Но давайте посмотрим, что произойдёт, если мы просто нажмём `Enter`.

РЕАГИРУЕМ НА ENTER

Нажатие на `Enter` в поле ввода (обратите внимание, оно у нас одно) приводит к отправке формы и при этом ещё «кликается» наша кнопка.

Т.е. фактически, делать нам ничего не нужно.

ИНТЕРАКТИВНОСТЬ

Добавим интерактивности и сделаем фильтрацию не по отправке формы, а по мере набора текста.

Для этого нам нужно событие `input`:

```
1 | filterTextEl.addEventListener('input', (event) => {  
2 |     console.log(event);  
3 | });
```

INPUTEVENT

event будет типа `InputEvent` и содержать будет:

- тип изменения `inputType`
- и данные изменения `data`

```
▼ InputEvent {isTrusted: true, data: null, isComposing: false, inputType:
  bubbles: true
  cancelBubble: false
  cancelable: false
  composed: true
  currentTarget: null
  data: null
  dataTransfer: null
  defaultPrevented: false
  detail: 0
  eventPhase: 0
  inputType: "insertFromPaste"
  isComposing: false
  isTrusted: true
  ▶ path: (8) [input, label, form, div, body, html, document, Window]
  returnValue: true
  sourceCapabilities: null
  ▶ srcElement: input
  ▶ target: input
  timeStamp: 65491.899999999362
  type: "input"
  view: null
  which: 0
  ▶ __proto__: InputEvent
```

INPUTEVENT

Причём срабатывать это событие будет в том числе на `Ctrl + v` и другие сочетания горячих клавиш. Но при этом в поле `data` не всегда будут содержаться нужные нам значения.

Поэтому для получения текущего значения в поле ввода всё равно придётся обращаться к `filterTextEl.value`.

PREVENT DEFAULT

Несмотря на то, что у события `input` нет поведения по умолчанию, давайте попробуем его отменить и посмотрим, что получится:

```
1 filterTextEl.addEventListener('input', (event) => {  
2   event.preventDefault();  
3   console.log(event.defaultPrevented); // всегда будет false  
4 });
```

PREVENT DEFAULT

На самом деле, не все события можно отменять: каждое событие содержит специальное свойство `cancelable`, которое и указывает на то, можно ли «отменить» событие:

Interface	<code>InputEvent</code>
Sync / Async	Sync
Bubbles	Yes
Cancelable	No
Composed	Yes
Target	<code>Element</code>
Default Action	None

ФИЛЬТРАЦИЯ

Таким образом, можем вынести функцию фильтрации в отдельный модуль и тестировать её отдельно, вызывая в обработчиках `input` и `click`:

```
1 export function filterByNameOrPhone(contacts, text) {  
2   return filterBy(contacts, o => (  
3     containsPhone(o.phone, text) || containsText(o.name, text)  
4   ));  
5 }
```

ВСЁ В СБОРЕ

```
1 filterBtnEl.addEventListener('click', (event) => {
2   event.preventDefault();
3   buildContactList(filterByNameOrPhone(data, filterTextEl.value), contactListEl, contactTag);
4 });
5
6 filterTextEl.addEventListener('input', () => {
7   buildContactList(filterByNameOrPhone(data, filterTextEl.value), contactListEl, contactTag);
8 });
```

ПРОСМОТР ДЕТАЛЕЙ ПОЛЬЗОВАТЕЛЯ

Добавим следующую функциональность: при клике на карточку пользователя должна показываться детальная информация о нём (по факту должен переключаться класс `hidden` у элемента с `data-section='details'`). Но кликать мы будем на элементе `li`, значит и обработчик должны повесить на него.

Но элементы `li` у нас генерируются динамически, это значит, что мы должны добавлять обработчики в цикле?

ДОБАВЛЕНИЕ ОБРАБОТЧИКОВ

В принципе, мы действительно, это можем сделать, важно здесь не увеличивать сильно сложности кода, чтобы не получилось:

```
1 function buildContactList(contacts, containerEl, elementTag) {
2   Array.from(containerEl.children).forEach(o => containerEl.removeChild(o));
3   contacts.forEach((o) => {
4     const el = document.createElement(elementTag);
5     ...
6     el.addEventListener('click', (event) => {
7       // а здесь ещё closure, в котором closure, в котором closure
8       console.log(event);
9     });
10    containerEl.appendChild(el);
11  });
12 }
```

CURRENTTARGET VS TARGET

В объекте события содержатся два поля:

1. `currentTarget` – объект, чей обработчик сейчас работает
2. `target` – объект, который был/будет в фазе `Target`

Если упрощённо, то `target` (в нашем случае), тот на ком действительно кликнули. И это может быть `span`, `img` и любой другой элемент, не обязательно `li`.

А `currentTarget` – это всегда тот, на ком сейчас работает обработчик.

Примечание*: при выводе через `console.log(event)` `currentTarget` будет показан как `null`. Но на самом деле он не `null`:

```
console.log(event);  
console.log(event.currentTarget);
```

CURRENTTARGET

В данном случае нам нужен будет именно `currentTarget`, потому что нам не важно, на каком элементе внутри `li` кликнули:

```
const contactDetailsEl = event.currentTarget.querySelector('[data-section=details]');
contactDetailsEl.classList.toggle('hidden');
```

ОТКУДА ВСЁ БЕРЁТСЯ?

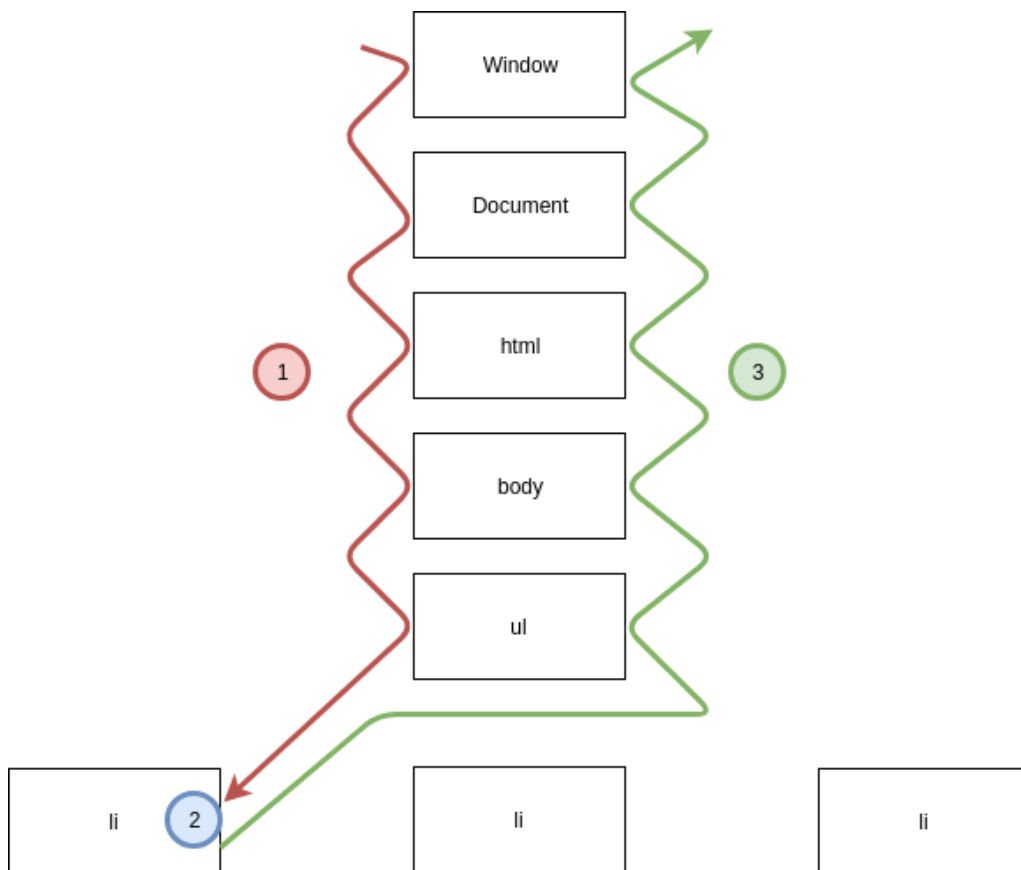
Почему мы кликаем на вложенных элементах `li`, а обработчик срабатывает на `li`?



CAPTURING & BUBBLING

CAPTURING И BUBBLING

На самом деле, возникающее событие проходит от объекта `window` до элемента, на котором возникает (`target`), через всех родителей, а затем возвращается вверх тем же путём:



1. Фаза Capturing
2. Фаза Target
3. Фаза Bubbling

BUBBLING ИЛИ CAPTURING?

Большинство `eventListener`’ов работают на фазе `Bubbling`, потому что она выставляется по умолчанию, когда вы добавляете `eventListener`.

Работать на фазе `Capturing` позволяет последняя опция в методе `addEventListener`:

- `true` <- рекомендуемый вариант
- `{capture: true}`

Примечание*: обратите внимание, что если вы добавляете обработчик на фазе `Capturing`, то и при удалении должны указать последний аргумент (`true`).

BUBBLE

Диаграмма, показанная на предыдущем слайде, в целом верна за одним исключением: не все события «всплывают».

Всплывает событие или нет, определяется флагом «Bubbles»:

Interface	<code>FocusEvent</code>
Sync / Async	Sync
Bubbles	No
Cancelable	No
Targets	<code>Window</code> , <code>Element</code>
Composed	Yes
Default Action	None

Здесь важно отметить, что сказано именно про всплытие.

`Capturing` будет по-прежнему работать.

STOPPROPAGATION

Остановить дальнейшее продвижение события по цепочке можно с использованием метода `stopPropagation`.

Но делать это не рекомендуется, т.к. в таком случае могут «отвалиться» другие скрипты, в первую очередь скрипты сборки аналитики вроде Google Analytics и Yandex Metrica.

EVENT DELEGATION

На базе этого возникла техника, которая называется

Event Delegation — мы можем делегировать обработку события общему предку, например, элементу `ul` и не навешивать обработчик на каждый элемент `li`.

```
1 | contactListEl.addEventListener('click', (event) => {  
2 |     console.log(event);  
3 | });
```

Но как теперь узнать, на каком конкретно элементе кликнули?

PARENT FINDING

`Element` содержит метод `closest`, который позволяет искать по CSS-селектору начиная от текущего элемента и выше (по родителям).

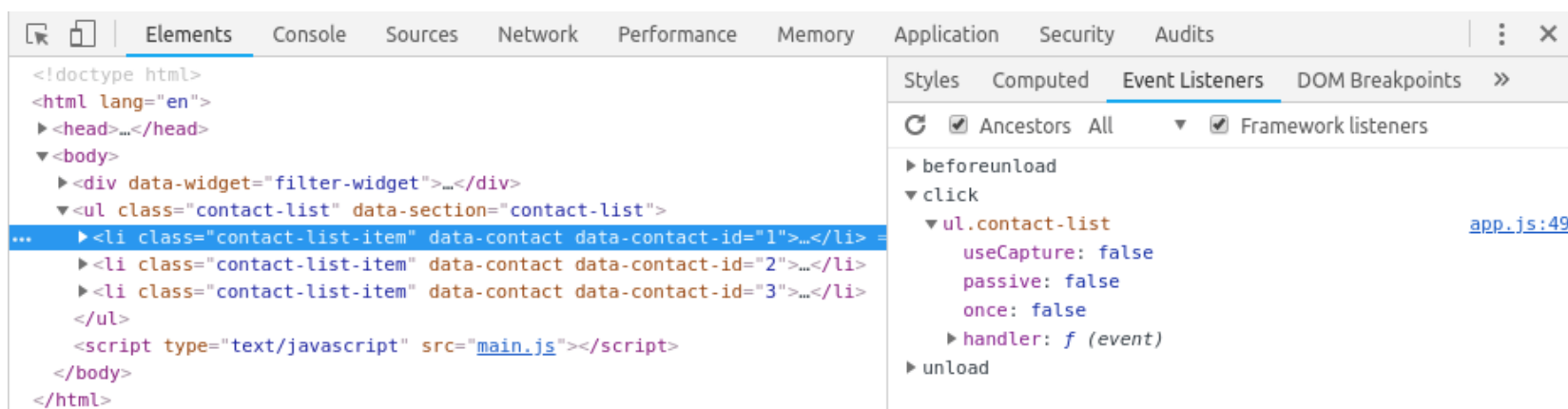
Но поддержка его хромает, поэтому реализацию мы оставляем на ваше усмотрение (можете подглядеть polyfill в MDN):

```
1  contactListEl.addEventListener('click', (event) => {  
2    const contactEl = event.target.closest('[data-contact]');  
3    const contactDetailsEl = contactEl.querySelector('[data-section=details]');  
4    contactDetailsEl.classList.toggle('hidden');  
5  });
```

`currentTarget` мы уже использовать не можем, потому что он всегда будет указывать на `ul`.

EVENTLISTENERS

Посмотреть все события, назначенные на элемент, можно с помощью вкладки **Event Listeners**, где опция **Ancestors** указывает, показывать ли eventListener'ы родителей:



ЗВОНОК КЛИЕНТУ

Осталась последняя функция в нашем импровизированном приложении — звонок клиенту.

Необходимо, чтобы при нажатии на ссылку «Звонок» открывалось какое-то модальное окно для звонка. Нам пока достаточно сделать перехват этого события.

Но возникает ряд вопросов. Если ссылка «Звонок» находится внутри `li`, то:

1. Все клики будут проходить через `ul`?
2. Снова добавлять обработчик события в цикле?

TARGET

Самый простой вариант — снова задействовать `target` и делегировать обработку события:

```
1  contactListEl.addEventListener('click', (event) => {
2    if (event.target.dataset.action === 'call') {
3      event.preventDefault();
4      // TODO: make call
5      return;
6    }
7    const contactEl = event.target.closest('[data-contact]');
8    const contactDetailsEl = contactEl.querySelector('[data-section=details]');
9    contactDetailsEl.classList.toggle('hidden');
10 });
```

Но тогда при увеличении функциональности карточки наш `eventListener` будет делать слишком много всего.



ТЕХНИЧЕСКИЙ ДОЛГ

Остаётся лишь аккуратно разделить наш основной скрипт на классы и модули (например, вынести фильтр в один класс, работу с контактами — в другой, а использовать их через связующий объект).

Не забывайте о том, что если вы пишете «один большой плоский модуль `app.js`», который делает всё, то рано или поздно он превратится в неподдерживаемый код.

ГЕНЕРАЦИЯ СОБЫТИЙ

Помимо перехвата уже существующих событий мы можем также программно генерировать события и отправлять их с помощью метода `dispatchEvent`. Но об этом мы поговорим в следующих лекциях.

ON*

Q: Стоп, стоп, но во многих статьях используются свойства вроде `onclick` вместо `addEventListener`

A: Да, действительно, такая возможность есть, в спецификации она называется `EventHandlers`

EVENTHANDLERS

<https://html.spec.whatwg.org/#event-handler-attributes>

Мы можем назначать объектам, перечисленным в спецификации, обработчики определённых событий через свойства `on*`:

```
1 filterBtnEl.onclick = (event) => {  
2   ...  
3 };
```

Указанные обработчики событий никак не конфликтуют с теми, что были добавлены через `addEventListener`, но имеют ряд особенностей:

1. Работают на этапе `Bubbling`
2. Можно навесить только один разработчик (он перетрёт предыдущие)
3. Чтобы удалить обработчик, достаточно записать:

```
filterBtnEl.onclick = null
```

EVENTHANDLERS

Обычно использование eventHandler'ов считается не очень хорошим решением, поскольку есть `addEventListener`.

Но вы часто можете встретить их использование в коде. Чаще всего их используют тогда, когда не предполагают «навешивать» более одного обработчика события.

ИТОГИ

Сегодня мы с вами рассмотрели достаточно много важных вещей, а именно:

- интерфейс `EventTarget`
- интерфейс `Event`
- фазы обработки событий
- Event Delegation



Спасибо за внимание!

Время задавать вопросы

АЛЕКСЕЙ ДАЦКОВ



alex.smap@gmail.com



[alex.smap](https://t.me/alex.smap)



[alex.smap](https://vk.com/alex.smap)