

Конспект по теме 2.2 «Работа с HTML-формами»

Содержание конспекта:

1. HTML-формы
2. События текстовых полей. Элементы `input "text"` и `textarea`

1. HTML-формы

HTML-форма — это раздел документа, позволяющий пользователю вводить информацию для последующей обработки системой.

Формы являются стандартом для получения данных от пользователя. Единственная альтернатива формам — это встроенные возможности браузера в виде диалоговых окон **Prompt** и **Confirm**, которые использовались нами ранее для взаимодействия с пользователем.

Функция **Prompt** ():

```
const years = prompt('Сколько вам лет?', 100);  
alert('Вам ' + years + ' лет!');
```

Функция **Confirm** ():

```
const isStudent = confirm("Вы студент?");  
alert(isStudent);
```

Недостатки Prompt ():

1. Синхронная работа функции: программа возобновит свою работу только после закрытия такого окна. Поведение диалоговых окон, встроенных в браузер, вызывает блокировку выполнения всего кода на JavaScript, в том числе таймеров, обработчиков событий, которые были назначены ранее и наступили, и всего последующего кода.
2. Невозможность стилизации: CSS-стили недоступны для элементов интерфейса браузера.
3. Задав дополнительную опцию в настройках браузера, пользователь вовсе может отключить показ диалоговых окон, что может сделать невозможной дальнейшую работу с вашим сайтом.
4. Политика производителей браузеров не рекомендует пользоваться данными методами.

Недостатки Confirm ()

У Confirm аналогичные недостатки, а также нельзя добавить варианты ответа кроме существующих, возвращающих результат true в случае положительного ответа и false в случае отрицательного.

Рассмотрим, каким образом можно взаимодействовать с HTML-формами посредством JavaScript.

Задача: форма регистрации — создать простую форму регистрации пользователя для сайта по управлению семейными финансами. Форма приведена на рисунке:

ФИО:
ФИО

E-mail:
E-mail

Выберите страну проживания:
Россия
Азербайджан
Армения
Белоруссия

Тип аккаунта:
☐ Личный ☐ Семейный ☒ Бизнес

☒ согласен на обработку персональных данных

Комментарии:
Комментарии

Зарегистрироваться

Как мы видим, в форме представлены следующие поля:

- ФИО, текстовое поле;
- Email, текстовое поле;
- Выберите страну проживания, select;
- Тип аккаунта, radio;
- Согласен на обработку персональных данных, checkbox;
- Комментарии, textarea;
- Кнопка «Зарегистрироваться».

В итоге, нажав на кнопку, мы должны отправить на сервер полученные от пользователя данные.

Таким образом, если вам получить данные от пользователя, выбор HTML-форм является очевидным решением в связи с их явным превосходством над встроенным функционалом браузера.

2. События текстовых полей. Элементы input “text” и textarea

Типы полей и их основные события приведены в таблице:

Типы полей	input text, textarea	select, radio, checkbox
События	input, change, focus, blur	change, (focus, blur)

Рассмотрим пример получения введённого значения из поля ФИО.

Данные, введённые в поле, получают в два этапа:

1. Получить ссылку на элемент поля ввода в дереве DOM.
2. Получить значение, введённое в это поле.

Первый пункт выполняется с помощью методов объекта document: `querySelector`, `getElementById` и т. д.

Второй пункт решается обращением к свойству `value` полученного узла.

Текстовое поле type="text". Чтение значения

HTML-разметка поля ФИО:

```
<label>
  ФИО: <input type="text" id="fio" name="fio">
</label>

<button id="registerButton">Зарегистрироваться</button>
```

Чтобы получить данных, введённые в текстовое поле, необходимо:

1. Получить ссылку на элемент поля ввода в дереве DOM.
2. Получить значение, введённое в поле.

Получить значение возможно, прочитав свойство `value` текстового поля.

Изменение значения текстового поля (JavaScript-код, в котором в консоль выводится ФИО):

```
const button = document.getElementById('registerButton');

button.addEventListener('click', e => {
  const name = document.getElementById('fio');
  const user = name.value;
  console.log(`Пользователь ${user} зарегистрирован`);
});
```

Для того, чтобы программно изменить значение текстового поля достаточно записать новое значение в свойство `value`.

Давайте представим, что нам необходимо, чтобы email пользователя при нажатии на кнопку «Зарегистрироваться» очищался от пробелов в начале и в конце строки. Для этого будем использовать метод `trim()`, описанный в 5 строке нашего примера. Он удаляет пробельные символы в начале и в конце строки, после чего мы записываем новое значение в поле e-mail.

```
const button = document.getElementById('registerButton');

button.addEventListener('click', e => {
  const email = document.getElementById('email');
  email.value = email.value.trim();
  console.log(`Пользователь ${email.value} зарегистрирован`);
});
```

Метод `trim()` в строке 5 удаляет пробельные символы с начала и конца строки, и мы записываем новое значение в поле e-mail.

События текстового поля `type="text"`

Наиболее часто используемые события текстового поля:

- `focus`
- `blur`
- `input`
- `change`

1. Событие `input`

На всех текстовых полях ввода событие `input` возникает каждый раз, когда мы вводим новый символ, удаляем символ или ещё как-то меняем введенное значение, в том числе с помощью вставки, вырезания и так далее.

В примере ниже при вводе значения в текстовое поле оно отображается в элементе **result**:

```
const fio = document.getElementById('fio');

fio.oninput = () =>
  document.getElementById('result').innerHTML = fio.value;
```

То же самое с использованием `addEventListener`:

```
onInput = (e) => document.getElementById('result').innerHTML = e.target.value;

fio.addEventListener("input", onInput);
```

2. Событие `change`

Есть также событие `change`, которое доступно и на текстовых полях, и на всех остальных: чекбоксы, радиокнопки, списки выбора. Событие `change` на текстовых полях возникает при потере фокуса.

В примере ниже при изменении значения и последующей потере фокуса в поле ФИО его значение отображается в элементе **result**:

```
const fio = document.getElementById('fio');

fio.onchange = () =>
  document.getElementById('result').innerHTML = fio.value;
```

3. События `focus`, `blur`

В проектировании интерфейсов есть понятие «фокус ввода». Пользователь одновременно может вводить данные только в одно поле. Считается, что поле находится в фокусе, если пользователь сейчас может вводить данные в него.

Признаком фокуса может служить наличие курсора в этом поле, а также изменение внешнего вида поля. При переходе к заполнению следующего поля у текущего поля фокус теряется. Именно фокус ввода переключается с помощью клавиши `Tab` в большинстве интерфейсов.

У элементов HTML-форм есть специализированные события для работы с фокусом:

- **focus** — когда поле ввода становится активным (в фокусе) с помощью клика мышки по полю или по метке поля либо при переключении фокуса клавишей `Tab`;
- **blur** — когда поле ввода теряет фокус.

Задача: вывод подсказки на поле при фокусе — необходимо показывать подсказку, как заполнять поле именно в тот момент, когда пользователь его заполняет.

HTML-разметка для нашего примера следующая:

```
<div>
  <label>E-mail:</label>
  <input type="text" placeholder="E-mail" name="e-mail" id="e-mail">
  <div class="hint hidden">Адрес электронной почты должен содержать "@"</div>
</div>
```

Подсказка в элементе `<div>` по умолчанию скрыта. И нам необходимо показать её, убрав класс **hidden** в тот момент, когда пользователь начнёт вводить сообщение, и скрывать, вернув **hidden**, когда пользователь закончит ввод.

Решение:

```
const email = document.getElementById('e-mail');
```

```
const hint = document.querySelector('.hint');

showHint() => hint.classList.remove('hidden');

hideHint() => hint.classList.add('hidden');

email.addEventListener('focus', showHint);
email.addEventListener('blur', hideHint);
```

Многострочное текстовое поле `textarea`

Запись и чтение из `textarea` производится аналогично тому, как это происходит для текстового поля.

События `textarea`

События `input`, `change`, `focus`, `blur` на поле `textarea` обрабатываются так же, как и для текстового поля.

Поле для вывода `output`

Поле `output` определяет не редактируемую для пользователя область, в которую выводится информация. Чтение и программная запись производится аналогично тому, как это происходит для текстового поля.

События `output`

Поскольку `output` не изменяемо, то и событий `input`, `change` оно не имеет. События `focus`, `blur` на поле `output` обрабатываются так же, как и для текстового поля.

Список `select`. Чтение значения

Перейдём к выбору пользователем страны проживания с формы регистрации.

HTML-разметка поля:

```
<label>
  Выберите страну проживания:
</label>
<select id="country">
  <option value="RUS" selected>Россия</option>
  <option value="AZE">Азербайджан</option>
  <option value="ARM">Армения</option>
  <option value="BLR">Белоруссия</option>
  <option value="KAZ">Казахстан</option>
  <option value="KGZ">Киргизия</option>
</select>
<button id="registerButton">Зарегистрироваться</button>
```

Обратите внимание, что для выбора значения по умолчанию используется атрибут `selected`.

Список select. Чтение значения

Рассмотрим JavaScript-код, в котором в консоль выводится выбранная пользователем страна проживания при её изменении (change):

```
const countryList = document.getElementById('country');

countryList.addEventListener('change', event => {
  console.log(countryList.value);
  // значение value выбранного элемента (RUS)
  console.log(countryList.selectedIndex);
  // порядковый номер выбранного элемента
  console.log(countryList.options[countryList.selectedIndex].text);
  // текст выбранной опции (Россия)
});
```

- Свойство **value**, как и в случае с текстовым полем, содержит значение свойства value выбранной пользователем опции. В примере RUS, AZE, ARM.
- Свойство **selectedIndex** показывает порядковый номер выбранной опции option, начиная с 0.
- Список элементов-опций доступен через **select.options**. Выбранные опции имеют свойство option.selected = true.
- Свойство **text** выбранной пользователем опции содержит его текст (в примере Россия, Азербайджан...), строка 6.

Аналогично мы могли бы получить выбранное значение и при нажатии на кнопку «Зарегистрироваться», как мы это делали в примере с получением ФИО и email.

Представим, что нам необходимо программно проставить значение в списке. По геолокации, доступной в браузере, мы можем предположить и проставить страну проживания по умолчанию.

Сделать это можно двумя способами: поставив значение select.value либо установив свойство select.selectedIndex в номер нужной опции:

```
const countryList = document.getElementById('country');
countryList.selectedIndex = 2; // по порядковому номеру с 0
countryList.value = "ARM"; // ИЛИ по значению value
```

Список select. Выбор нескольких значений

При помощи атрибута **multiple** можно создать список с возможностью множественного выбора. На современных сайтах, веб-приложениях этот вид HTML-элемента практически не встречается. Как правило, если необходим список с возможностью выбора нескольких значений, используют различные JavaScript-компоненты, совместимые с используемыми в проекте библиотекой, фреймворком: React, Angular,

Vue и др. Их преимущества: кастомизация внешнего вида компонента, а также возможны дополнительные функции типа поиска по всем элементам. Для чтения выбранных элементов и их записи в таком списке приходится работать с массивом, что несколько сложнее. При желании вы можете разобраться со списком самостоятельно, см. ссылку из доп. материалов 2, [Свойства и методы формы](#).

На списках, а также чекбоксах и радиокнопках, событие change возникает при выборе нового значения. Обработывается оно так же, как и для текстового поля.

Рассмотрим пример для списка с возможностью выбора одного значения:

```
const country = document.getElementById('country');

country.onchange = () =>
  document.getElementById('result').innerHTML = country.value;
```

События focus, blur для этого типа поля не являются такими популярными для обработки, как для input text. Обычно достаточно обработки change.

HTML-разметка поля:

```
<label>
  Выберите способ оплаты:
</label>
<select id="payMethod">
  <option value="cardSite" selected>Картой на сайте</option>
  <option value="cardDelivery">Картой курьеру</option>
  <option value="cashDelivery">Наличными курьеру</option>
</select>
<button id="send">ОК</button>
```

```
const payMethods = document.getElementById('payMethod');

payMethods.addEventListener('change', event => {
  const { value, options, selectedIndex } = event.currentTarget;
  console.log(value);
  console.log(options[selectedIndex].text);
});
```

При клике на «Картой курьеру» в строках 5 и 6 будет выведено cardDelivery и Картой курьеру.

Обратите внимание, как внутри обработчика событий используется event .

Радио-группа radio. Чтение значения

Перейдём к выбору пользователем типа аккаунта с формы регистрации.

HTML-разметка поля:

Тип аккаунта:

```
<label>
  <input type="radio" name="type" value="Личный" checked id="private">
  Личный
</label>
<label>
  <input type="radio" name="type" value="Семейный" id="family">
  Семейный
</label>
<label>
  <input type="radio" name="type" value="Бизнес" id="business">
  Бизнес
</label>
```

Обратите внимание, что если элементы имеют один и тот же **name**, то они считаются группой радио-кнопок. Иначе можно будет выбирать каждую радио-кнопку по отдельности, и при выборе одной не будет сбрасываться выбор с других. Для выбора значения по умолчанию используется атрибут **checked**.

JavaScript-код, в котором в консоль выводится выбранный пользователем тип:

```
const button = document.getElementById('registerButton');

button.addEventListener('click', e => {
  const typeRadios = document.getElementsByName('type');
  for (let i=0; i < typeRadios.length; i++) {
    if (typeRadios[i].checked) {
      console.log(`Выбран тип ${typeRadios[i].value}`);
    }
  }
});
```

Радио-группа radio. Запись значения

Самый простой способ для программной простановки значения группы радио-кнопок — использовать ID элемента:

```
document.getElementById("family").checked = true;
```

События radio

На радиокнопках событие change возникает при выборе нового значения. Рассмотрим пример, в котором при изменении значения оно выводится в элемент result:

```
const typeRadios = document.getElementsByName('type');

for (var i = 0; i < typeRadios.length; i++) {
  typeRadios[i].addEventListener('change', (evt) => {
    const { value } = evt.target;
    document.getElementById('result').innerHTML = value;
  });
}
```

События focus, blur для этого типа поля не являются такими популярными для обработки, как для input text. Обычно достаточно обработки change.

Чекбокс (checkbox). Чтение значения

Перейдём к обработке чекбокса «Согласен на обработку персональных данных» с формы регистрации.

HTML-разметка поля:

```
<label>
  <input type="checkbox" name="isAgree" id="isAgree" checked> согласен на обработку
  персональных данных
</label>
```

Для выбора значения по умолчанию используется атрибут checked.

Для проверки, установлен ли чекбокс, используется свойство checked.

```
const checkbox = document.getElementById("isAgree");
console.log(checkbox.checked); // true / false
```

У чекбоксов определен CSS-псевдокласс для неопределённого состояния, :indeterminate. Предлагаем ознакомиться с ним самостоятельно, см. ссылку из доп. материалов 3. [:indeterminate](#)

Чекбокс (checkbox). Запись значения

Для программной простановки значения чекбокса можно использовать следующий код:

```
document.getElementById("isAgree").checked = true; // или false
```

События checkbox

На чекбоксах событие change возникает при выборе нового значения. Рассмотрим пример, в котором при изменении состояния оно выводится в элемент result:

```
const isAgree = document.getElementById('isAgree');  
  
isAgree.onchange = () => document.getElementById('result').innerHTML = isAgree.checked;  
// true / false
```

События focus, blur для этого типа поля не являются такими популярными для обработки, как для input text. Обычно достаточно обработки change.

Атрибут disabled

При помощи атрибута **disabled** можно заблокировать поля формы разных типов, чтобы они были недоступны для изменения пользователем. При этом значение в поле всё ещё можно считать.

Пример отключённого текстового поля:

```
<label>  
  ФИО: <input type="text" id="fio" disabled>  
</label>
```

При помощи следующего кода можно проверить, является ли элемент отключённым:

```
console.log(document.getElementById(fio).disabled);  
// true или false
```

При помощи следующего кода можно проставить атрибут **disabled**:

```
document.getElementById(fio).disabled = true  
// или false
```

Типы полей и события

Типы полей	input text, textarea	select, radio, checkbox
События	input, change, focus, blur	change, (focus, blur)

Формы HTML-страницы: document.forms

К конкретной форме на странице можно получить доступ несколькими способами:

1. Через её **id**:

```
const form = document.getElementById('register-form');
```

2. Через **document.forms**. Если форме добавить атрибут **name**, то к ней можно получить доступ через свойство **forms** элемента **document**, в котором хранится коллекция всех форм на странице:

```
<form name="register-form">
  <!-- ... -->
</form>
```

```
const form = document.forms['register-form'];
// ...
```

Отправка формы на сервер, submit

В итоге у нас получилась форма регистрации со следующей разметкой (представлена в сокращённом виде):

```
<form name="register-form" method="get" action="">
  <div>
    <label>ФИО:</label>
    <input type="text" placeholder="ФИО" name="fio" id="fio">
  </div>
  <div>
    <label>Выберите местонахождение:</label>
    <select name="country" id="country">
      <option value="RUS">Россия</option>
      <!-- ... -->
    </select>
  </div>
  <!-- ещё поля -->
  <div>
    <label><input type="checkbox" name="isAgree" id="isAgree" checked> согласен на
    обработку персональных данных </label>
  </div>
  <button name="register-button" id="register-button"
  type="submit">Зарегистрироваться</button>
</form>
```

Обработаем отправку данных:

```
const form = document.forms['register-form'];
```

```
form.addEventListener('submit', event => {
  // тут может быть обработка данных до отправки формы
});
```

При отправке формы на сервер обратите внимание:

1. Чтобы отработало событие **submit** формы, **button** должен иметь **type="submit"** или тип должен отсутствовать — у кнопки тип **submit** по умолчанию. Если же у кнопки другой тип, например, **button**, то форму всё равно можно засабмитить вручную:

```
registerButton.addEventListener('click', e => {
  // тут может быть обработка данных до отправки формы
  form.submit(); // сабмитим так, если у кнопки type="button"
});
```

2. Форме нужно указать **action** — это URI программы на сервере, которая будет обрабатывать запрос и возвращать ответ. В нашем случае он пустой.

3. Форме можно указать метод, которым будет происходить отправка на сервер. В нашем примере `method="get"`. Может принимать значения **get** / **post** — соответствуют одноимённым HTTP-методам. В случае **get** данные из формы добавляются к URI (Uniform Resource Identifier) атрибута **action**, их разделяет **?**, и полученный URI посылается на сервер — можем его видеть в адресной строке браузера. В случае **post** данные из формы включаются в тело формы и посылаются на сервер.

4. После отправки формы страница перезагружается.

Обработка формы на клиентской стороне

А что, если нам не нужно отправлять форму на сервер для обработки? Или мы хотим, чтобы данные формы передавались без перезагрузки страницы, при помощи AJAX?

Тогда нужно изменить наш пример следующим образом:

- `button` должен иметь `type="button"`;
- убираем атрибуты формы `action` и `method`;
- обрабатываем событие нажатия на кнопку, а не `submit` формы.

Если по какой-то причине кнопка имеет тип `submit`, то можно добиться такого же результата как у кода выше, отменив в обработке события **submit** действия браузера по умолчанию:

```
const form = document.forms['register-form'];

button.addEventListener('click', e => {
  // тут может быть отправка формы через AJAX
  const fio = document.getElementById('fio');
  console.log(`Пользователь ${fio} зарегистрирован`);
});
```

Сброс формы reset

Помимо события **submit** на форме наступает событие **reset**, которое браузер тоже обрабатывает сам. Он возвращает форму в то состояние, которое задано в HTML разметке изначально. Допустим, у нас в форме на сервере уже задано имя пользователя:

```
<form name="register-form" method="get" action="">
  <div>
    <label>ФИО:</label>
    <input type="text" placeholder="ФИО" name="fio" value="Василий">
  </div>
  <div>
    <label>E-mail:</label>
    <input type="text" placeholder="E-mail" name="e-mail">
  </div>
  <!-- ещё поля -->
  <button name="register-button" type="submit">Зарегистрироваться</button>
</form>
```

Дальше мы поменяем имя и email. После наступления события **reset** поле email будет очищено, а вот поле fio будет сброшено и будет содержать имя Василий, т. е. сброс — это именно возврат к исходному состоянию, а не просто очистка.

Событие **reset** возникает в следующих случаях:

- нажатие на кнопку `<input type="reset">` или `<button type="reset"></button>`;
- вызов метода `reset` у элемента формы.

Нам как раз подходит второй вариант. Воспользуемся встроенным методом `reset` формы:

```
document.forms['register-form'].addEventListener('submit', event => {
  event.preventDefault();
  // обработка полей формы, передача на сервер при необходимости
  const form = event.currentTarget;
  form.reset();
});
```

Валидация полей форм при помощи `checkValidity`

Стандартная валидация полей `checkValidity`

С помощью метода `checkValidity`, который является частью JavaScript Validation API, можно проверять текстовые поля на соответствия ограничениям и выводить соответствующее сообщение.

Ограничения могут быть следующими:

- `minlength` — минимальная длина значения в символах;
- `maxlength` — максимальная длина значения в символах;
- `min` — минимальное значение для `input type="number"` ;
- `max` — максимальное значение для `input type="number"`;
- `required` — является ли поле обязательным для заполнения;
- `pattern` — проверка значения на соответствие регулярному выражению.

Подробнее с Validation API предлагаем ознакомиться самостоятельно, см. ссылку из доп. материалов 4. [Руководство по HTML-формам](#).

Рассмотрим пример, в котором проставим минимальную и максимальную длину для поля E-mail (5-30 символов), провалидируем введенное значение после нажатия на кнопку «Проверить» и выведем соответствующее сообщение:

```
<input id="email" minlength="5" maxlength="30">
<button onclick="validate()">Проверить</button>
<p id="demo"></p>
```

```
const validate = () => {
  let txt = "";
  if (!document.getElementById("email").checkValidity()) {
    txt = "E-mail должен быть длиной от 5 до 30 символов";
  } else {
    txt = "Валидация прошла успешно";
  }
  document.getElementById("demo").innerHTML = txt;
}
```

Итоги

- Для сабмита формы используется событие `submit`;
- Для сброса формы к начальному состоянию используется событие `reset`;
- С помощью метода `checkValidity`, который является частью JavaScript Validation API, можно проверять текстовые поля на соответствия ограничениям.

Итоги по теме

- Основными элементами форм являются:
 1. input text,
 2. select,
 3. radio,
 4. checkbox,
 5. textarea,
 6. output.
- Основными событиями полей форм являются input, change, focus, blur;
- Событие формы submit используется при отправке формы на сервер,
- reset — для сброса формы к начальному состоянию;
- К форме на странице можно обращаться с помощью document.forms;
- Производить валидацию формы можно при помощи checkValidity.

Дополнительные материалы:

1. [Chromium policy on JavaScript dialogs](#)
2. [Свойства и методы формы](#)
3. [:indeterminate](#)
4. [Руководство по HTML-формам](#)

Материалы, использованные при подготовке:

- [<input>: The Input \(Form Input\) element](#)
- [<form>](#)
- [<select>: The HTML Select element](#)
- [<textarea>](#)
- [<input type="checkbox">](#)
- [<input type="radio">](#)
- [JavaScript Validation API](#)