

FORMS & METRICS



АЛЕКСАНДР ШЛЕЙКО



АЛЕКСАНДР ШЛЕЙКО

Программист в Яндекс



a.shleyko@yandex.ru



vk.com/shleiko



[@dustyo_0](https://t.me/@dustyo_0)



ПЛАН ЗАНЯТИЯ

1. [Валидация](#)
2. [Position & Metrics](#)
3. [Кэширование данных в LocalStorage](#)
4. [Input vs Change](#)



ВАЛИДАЦИЯ



ЗАДАЧА

Перед нами поставили задачу: реализовать форму регистрации на сайте. Давайте проанализируем, как это сделано у локомотивов индустрии, для этого возьмём Yandex и Mail.ru

ПРИМЕР РЕАЛИЗАЦИИ


Регистрация

Имя

Фамилия

Придумайте логин

Придумайте пароль



Повторите пароль

Номер мобильного телефона

[У меня нет телефона](#)

Зарегистрироваться

Яндекс Почта

ВАЛИДАЦИЯ

Получать значения из полей ввода мы уже умеем, ключевых вопросов при реализации остаётся несколько:

1. Валидация — когда показывать сообщение об ошибке, когда скрывать?
2. Какие типы полей использовать для ввода телефона и даты рождения?
3. Как «запоминать» ввод пользователя, чтобы, если он обновит случайно страницу, данные не потерялись?

К форме регистрации это отношения, конечно, имеет мало, но есть вопрос, связанный с правильной очисткой формы.

ПО ПОРЯДКУ

Итак, начнём с форм. Для форм у нас есть события:

- `submit`
- `reset`

И следующие ключевые методы:

- `submit`
- `reset`
- `checkValidity`
- `reportValidity`

CAN I USE

#

Constraint Validation API - LS

Usage

% of all users

?

Global

87.47% + 5.32% = 92.79%

API for better control over form field validation.
Includes support for `checkValidity()`,
`setCustomValidity()`, `reportValidity()` and
validation states.

Current aligned	Usage relative	Date relative	Apply filters	Show all	?				
IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Blackberry Browser
		2-3.6							
		^{1 2 3} 4-28	4-14	3.1-5	10-11.5	3.2-4.3			
	^{1 2 3} 12-13	^{1 2} 29-48	^{1 2 3} 15-24	^{1 2 3} 5.1-7	^{1 2 3} 12.1	^{1 2 3} 5-6.1		2.1-3	
6-9	^{1 2} 14-16	² 49-50	^{1 2} 25-39	^{1 2} 7.1-9.1	^{1 2} 15-26	^{1 2} 7-9.3		^{1 2 3} 4-4.3	
^{1 2 3} 10	17	51-64	40-71	10-11.1	27-57	10-11.4		^{1 2} 4.4-4.4.4	7
^{1 2 3} 11	18	65	72	12	58	12.1	all	67	^{1 2} 10
		66-67	73-75	12.1-TP		12.2			

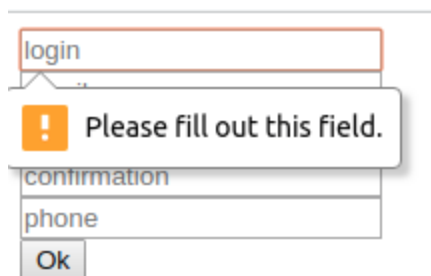
СОБЫТИЯ

- `submit` возникает при попытке отправки формы через кнопку с типом `submit` или через вызов метода `submit` на форме;
- `reset` возникает при клике на кнопку с типом `reset` или через вызов метода `reset` на форме.

ОГРАНИЧЕНИЯ

Поля `input` и другие элементы ввода позволяют с помощью атрибутов устанавливать ограничения на ввод, что приводит к эффектам:

1. Браузер проводит валидацию полей.
2. Если поле не валидно, выставляется псевдо-класс `:invalid` (`:valid` в противном случае).
3. Если хотя бы одно поле не валидно, событие `submit` не генерируется, форма не отправляется и устанавливается фокус на невалидном поле с выпадающей ошибкой:



The image shows a web form with three input fields: 'login', 'confirmation', and 'phone'. The 'login' field is highlighted with a red border, indicating it is invalid. A tooltip with an orange exclamation mark icon and the text 'Please fill out this field.' is displayed over the 'login' field. Below the 'phone' field is an 'Ok' button.

АТТРИБУТЫ

Список атрибутов, позволяющих установить ограничения можно найти в [спецификации HTML](#).

Кроме того, ряд значений атрибута `type` так же накладывает свои ограничения (например, `email`).

СТИЛИЗАЦИЯ

Мы можем стилизовать сами поля ввода через псевдо-классы, но обратите внимание, что, если выставить...:

```
input:invalid {  
  border: 1px solid red;  
}
```

...то поля будут сразу подсвечены красным (даже если пользователь их ещё не заполнял).

BROWSER ISSUES

В разных браузерах будет отличаться не только поведение, но и внешний вид сообщений, а также их текст.

Естественно, в коммерческой разработке почти никого не устраивает различное поведение в разных браузерах, все хотят «чтобы во всех браузерах было одинаково» `~_(\ツ)_/~`

Давайте, посмотрим, что мы можем сделать.

ОТКЛЮЧЕНИЕ НАТИВНОЙ ПРОВЕРКИ

Для отключения нативной проверки с прерыванием отправки выставим атрибут `novalidate` у формы (либо свойство `noValidate` через JS).

Теперь событие `submit` генерируется и мы можем его перехватить.

OLD STYLE

Для проверки самой формы мы можем последовательно вычитывать все поля из `form.elements` и применять к ним проверки на JS.

Но есть ли какая-то возможность совместить те ограничения, которые мы можем установить через атрибуты полей и JS?



CONSTRAINTS VALIDATION API

В спецификации определено [Contstraints Validation API](#).

ПРОВЕРКА САМОЙ ФОРМЫ

У самой формы тоже выставляется псевдо-класс `:invalid`, но нам это не сильно помогает.

Для проверки на то, валидна сама форма или нет, мы можем использовать метод `checkValidity`:

- возвращает `true`, если форма валидна (т.е. валидны все поля, входящие в неё);
- возвращает `false`, если форма не валидна (т.е. не валидно хотя бы одно поле, входящее в неё).

ПРОВЕРКА САМОЙ ФОРМЫ

```
1 form.addEventListener('submit', (evt) => {  
2   evt.preventDefault();  
3   const isValid = evt.currentTarget.checkValidity();  
4   if (!isValid) {  
5     // TODO: focus on first invalid field  
6   }  
7 });
```

reportValidity

Стоит отметить, что мы по-прежнему можем вызвать нативное сообщение браузера об ошибке с установкой фокуса на поле с помощью метода `reportValidity`.

Поиск невалидного поля

Решение с `reportValidity` нас не устраивает, а `checkValidity` не даёт информации о том, какое поле невалидно.

Попробуем перебрать все поля формы:

```
const first = [...form.elements].find(o => !o.validity.valid);
```

`form.elements` – `HTMLFormControlsCollection`

VALIDITYSTATE

Свойство `validity` содержит live-объект (о таких объектах мы говорили в предыдущих лекциях), который удовлетворяет следующему интерфейсу:

```
interface ValidityState {  
  readonly attribute boolean valueMissing;  
  readonly attribute boolean typeMismatch;  
  readonly attribute boolean patternMismatch;  
  readonly attribute boolean tooLong;  
  readonly attribute boolean tooShort;  
  readonly attribute boolean rangeUnderflow;  
  readonly attribute boolean rangeOverflow;  
  readonly attribute boolean stepMismatch;  
  readonly attribute boolean badInput;  
  readonly attribute boolean customError;  
  readonly attribute boolean valid; // <- true, если поле валидно  
};
```



ИТОГО

Итого мы с вами нашли первое невалидное поле, осталось научиться ставить на нём фокус и отображать ошибку.



POSITION & METRICS

КЛЮЧЕВЫЕ СОБЫТИЯ

Для полей ввода, а так же выпадающих списков `select` и областей для ввода `textarea` ключевыми являются следующие события:

- `focus` / `blur` (`focusin` / `focusout`)
- `change`
- `input`

Кроме того, у них есть методы:

- `blur`
- `click`
- `focus`
- `setCustomValidity` – установка кастомного сообщения валидации
- `checkValidity`
- `reportValidity`

УСТАНОВКА ФОКУСА

```
const first = [...form.elements].find(o => !o.validity.valid);  
first.focus();
```

ОТОБРАЖЕНИЕ СООБЩЕНИЯ

Сообщение об ошибке мы можем передавать либо через `data-*`, либо любым иным механизмом (вплоть до хранения их в массиве) и отображения исходя из `ValidityState`.

Ключевой вопрос, как научиться отображать сообщения подобные этому:

Регистрация

Имя

Пожалуйста, укажите имя

ТЕХНИКИ

Существует достаточно много техник, мы рассмотрим две ключевые:

1. CSS — благодаря псевдо-классу `:invalid`, а также разработанному нами методу JS, мы можем просто отпозиционировать конкретный элемент средствами CSS и отображать и скрывать его (что средствами JS, что средствами CSS).
2. Расчёт геометрии элементов (их размеров) и позиционирование всплывающего сообщения средствами JS (путём расчёта нужного отступа).

Первый способ достаточно простой и не должен вызвать у вас сложности, мы же займёмся вторым.

POSITION

Чтобы правильно научиться позиционировать элемент, нужно вспомнить инструменты, которые для этого есть в CSS, мы рассмотрим именно свойство `position`, а конкретно его значение `absolute`.

position: absolute

В спецификации для этого случая описано следующее:



In the absolute positioning model, a box is explicitly offset with respect to its containing block. It is removed from the normal flow entirely (it has no impact on later siblings). An absolutely positioned box establishes a new containing block for normal flow children and absolutely (but not fixed) positioned descendants.



Перевод:

В модели абсолютного позиционирования, элемент явно смещён относительно содержащего его блока. Он (элемент) удаляется из нормального потока (не влияет на последующие соседние элементы). Абсолютно позиционированный блок устанавливает новый содержащий блок для дочерних элементов нормального потока и абсолютно (но не фиксированных) потомков.

Containing Block

Containing Block для абсолютно позиционированного элемента определяется ближайшим родителем, у которого `position` не равен `static`, если такого нет, то доходим до `root` *.

* Схема дана с некоторыми упрощениями.

RELATIVE VS BODY

Отсюда вытекают две техники (первая из них фактически является CSS-схемой, но с расчётом геометрии):

1. Сделать некий контейнер с `position: relative;`, внутрь которого подставлять сообщение об ошибке с `position: absolute;` и правильно позиционироваться с использованием `top`, `left` (либо других свойств).
2. Добавлять элемент с ошибкой непосредственно в `document.body` и позиционироваться исходя из текущего положения на странице.

Мы рассмотрим оба варианта, начнём с первого.

КОНТЕЙНЕР С position: relative

```
1 <div class="form-control">
2   <input type="text" placeholder="login" required>
3 </div>
```

```
1 .form-control {
2   position: relative;
3 }
4 .form-input {
5   margin: 5px 10px;
6 }
7 .form-error {
8   position: absolute;
9   padding: 5px 10px;
10  width: 200px;
11  background: #fff;
12  box-shadow: 0 5px 20px 0 rgba(0, 0, 0, 0.1);
13  z-index: 999;
14 }
```



ПРАВИЛА ПОЗИЦИОНИРОВАНИЯ

Нам нужно отпозиционировать создаваемый нами элемент следующим образом:

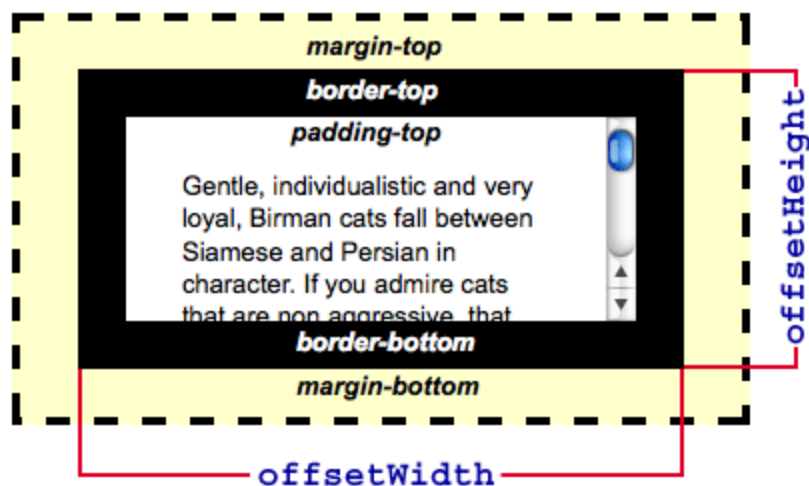
1. Левый край нашего элемента должен примыкать к правому краю поля ввода.
2. Центр (по вертикали) нашего элемента должен соответствовать центру поля ввода.

offset*

Свойства группы `offset*` дают нам следующую информацию:

- `offsetParent` – ссылка на ближайшего позиционированного родителя (`null`, если `display: none`);
- `offsetTop` – расстояние до `top` `offsetParent` 'а;
- `offsetLeft` – расстояние до `left` `offsetParent` 'а;
- `offsetWidth` – ширина элемента (включая `border`, `padding` и вертикальный скролл);
- `offsetHeight` – высота элемента (включая `border`, `padding` и горизонтальный скролл).

offset*



Изображение с сайта MDN

РЕШЕНИЕ

Таким образом, мы можем взять `offsetParent` у нашего поля ввода, сделать для него `appendChild` блока с ошибкой и выставить CSS-свойства `top` и `left`.

Вопрос к аудитории: почему не `offsetTop` и `offsetLeft`?

РАБОТА С CSS ИЗ JS

Несмотря на то, что в большинстве случаев рекомендуется через JS выставлять только CSS-классы, в нашем случае нужно выставить именно конкретные значения.

Общий синтаксис:

```
first.style.<propertyName> = <value>
```

Ключевых момента два:

1. Свойства с дефисом в CSS преобразуются в camelCase в JS.
2. Нужно указывать размерность величин, если она подразумевается (в нашем случае px)

ГЕОМЕТРИЯ

Вопрос к аудитории: какова должна быть формула расчёта положения?

ГЕОМЕТРИЯ

```
1  const first = [...form.elements].find(o => !o.validity.valid);
2  first.focus();
3  const error = document.createElement('div');
4  error.className = 'form-error';
5  error.textContent = 'Большой текст сообщения об ошибке
6                      на основании ValidityState';
7  first.offsetParent.appendChild(error);
8  error.style.top = `${first.offsetTop + first.offsetHeight / 2
9                      - error.offsetHeight / 2}px`;
10 error.style.left = `${first.offsetLeft + first.offsetWidth}px`;
```

Естественно, это нужно отрефакторить и вынести в отдельную функцию.

РЕЗУЛЬТАТ

login	
email	
password	
confirmation	
phone	
Ok	Reset

Большой текст сообщения об ошибке на основании
ValidityState

БЕЗ КОНТЕЙНЕРА

Без контейнера мы можем воспользоваться `document.body` в качестве контейнера, добавляя элемент туда, при этом методику расчёта придётся изменить:

1. Нам нужно узнать, насколько наш элемент отстоит от верхнего левого угла документа.
2. К этим значениям прибавить ширину и высоту элемента.

getBoundingClientRect

Возвращает координаты элемента относительно окна просмотра `viewport` (к сожалению, нет отдельного метода для получения координат относительно документа).

Вопрос к аудитории: можно ли справиться силами только `getBoundingClientRect` ?

position: fixed

На самом деле можно установить `position: fixed;` и туда приписать координаты `getBoundingClientRect`, но тогда при прокрутке страницы поле ввода будет прокручиваться, а сообщение об ошибке «зафиксируется» в положении.

Это удобно для всяких всплывающих уведомлений и модальных окон, но не в нашем случае.

ВЫЧИСЛЯЕМ КООРДИНАТЫ ЭЛЕМЕНТА

Координаты элемента относительно документа:

```
1  const first = [...form.elements].find(o => !o.validity.valid);
2  first.focus();
3  const error = document.createElement('div');
4  error.className = 'form-error';
5  error.textContent = 'Большой текст сообщения об ошибке
6                      на основании ValidityState';
7  document.body.appendChild(error);
8  const { top, left } = first.getBoundingClientRect();
9  error.style.top = `${window.scrollY + top + first.offsetHeight / 2
10                    - error.offsetHeight / 2}px`;
11 error.style.left = `${window.scrollX + left + first.offsetWidth}px`;
```

- `window.scrollX`, `window.scrollY` – позиция scroll'a;
- `top`, `left` – позиция элемента относительно viewport'a.

ПОКАЗ И СКРЫТИЕ ОШИБКИ

Хорошо, пока мы умеем показывать ошибку при `submit`, но теперь есть два других вопроса:

1. Что если мы хотим добавить валидацию по мере ввода?
2. Как выяснить, когда скрывать сообщение об ошибке?

К сожалению, в стандарте определено событие `invalid` (а события `valid` нет), да и возникает оно тогда, когда форма валидируется перед отправкой.

КЛЮЧЕВЫЕ СОБЫТИЯ

Итак, у нас есть события:

- `focus` / `blur` (`focusin` / `focusout`)
- `change`
- `input`

Обычно, ошибку отображают при событии `blur` (т.е. пользователь закончил редактировать поле) и скрывают при `focus`. Отличие от этих событий от тех, что в скобках, `blur` и `focus` не всплывают.

Таким образом, нам остаётся подписаться на эти события и реализовать показ/скрытие ошибок.



КЭШИРОВАНИЕ ДАННЫХ В `LocalStorage`



КЭШИРОВАНИЕ ДАННЫХ

Мы можем кэшировать данные полей в локальном хранилище браузера (разумеется, пароль кэшировать не стоит).

Что это за локальное хранилище?

Storage

Интерфейс, реализуемый объектами `localStorage` и `sessionStorage`, предоставляется нам браузером.

Представляет из себя key-value хранилище, ограниченные определённым `Origin`.

Origin

Origin – это набор из схемы, хоста (или домена) и порта.

Эта тройка определяет ограничения безопасности, не позволяющие скриптам с другого **Origin** получать доступ к некоторым объектам (в частности к локальному хранилищу).

Storage

```
1 interface Storage {  
2     readonly attribute unsigned long length;  
3     DOMString? key(unsigned long index);  
4     getter DOMString? getItem(DOMString key);  
5     setter void setItem(DOMString key, DOMString value);  
6     deleter void removeItem(DOMString key);  
7     void clear();  
8 };
```

Storage

Ключевое, что данный объект работает со строками: т.е. хранить он умеет только строковые ключи и строковые значения.

Поэтому для работы со `Storage` и хранения в нём объектов используется связка из `JSON.stringify()` / `JSON.parse`.

Вопрос к аудитории: что будет, если попытаться использовать объект в строковом контексте?



ЗАПИСЬ И ЧТЕНИЕ ДАННЫХ

В зависимости от стратегии данные можно сохранять перед перезагрузкой страницы или по мере ввода. И загружать при запуске скрипта.

ЗАПИСЬ ДАННЫХ

Данные с полей форм мы можем собрать через `form.elements` и преобразовать в объект, где ключами будут либо `data-*`, описывающие наши поля, либо `id`, либо `name`, либо `class`.

Пример для `id`:

```
const fields = [...form.elements].map(({ id, value }) => ({ id, value }));  
localStorage.setItem('fields', JSON.stringify(fields));
```

ЗАПИСЫВАЕМ ПРИ ОБНОВЛЕНИИ

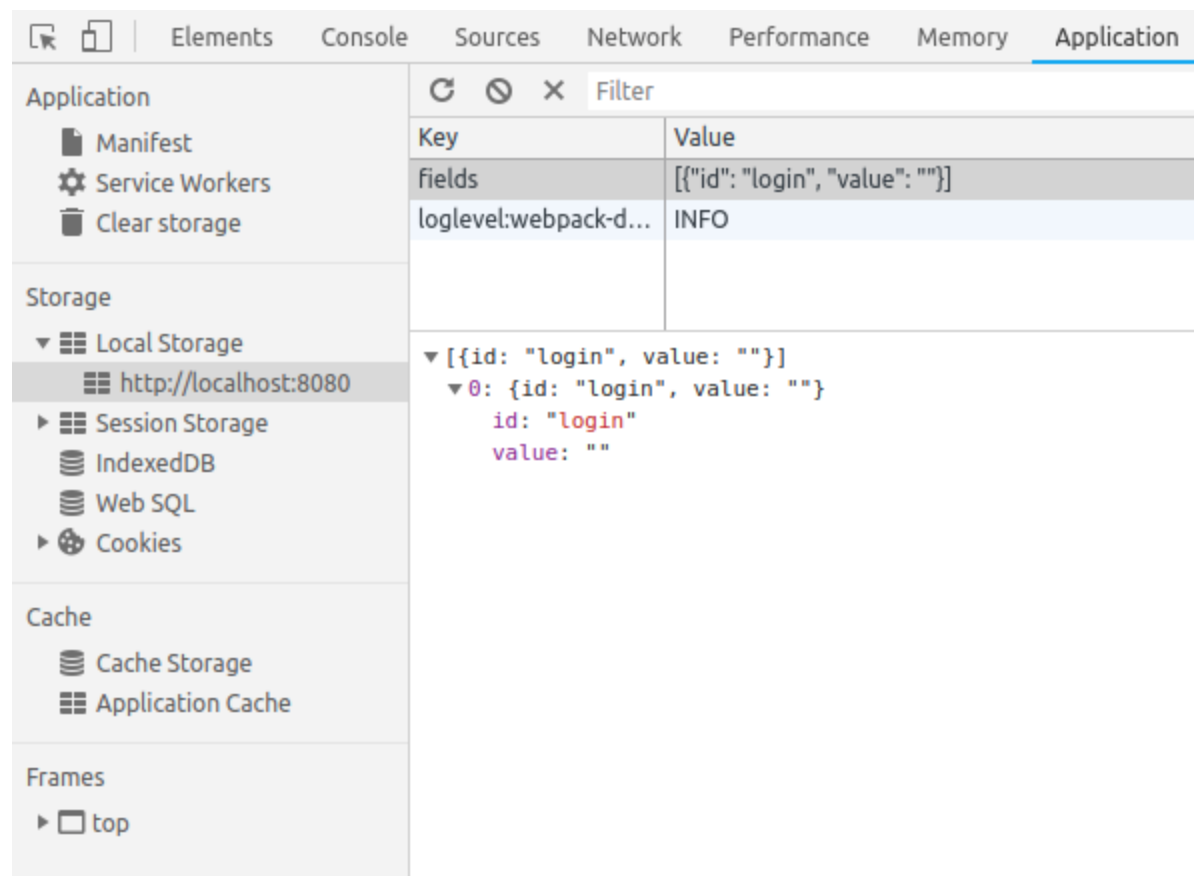
```
window.addEventListener('beforeunload', () => {  
  // TODO: save items here  
});
```


ЧИТАЕМ ПРИ ЗАГРУЗКЕ

При загрузке достаточно запускать:

```
1  try{
2    const fields = JSON.parse(localStorage.getItem('fields'))
3    if (fields !== null) {
4      // TODO: fill fields
5    }
6  } catch (e) {
7    console.error(e);
8  }
```

DEVELOPER TOOLS



Не стоит обманываться тем, что в DevTools объект показан как массив (на самом деле там строка).

SessionStorage VS LocalStorage

Функционально аналогична `localStorage` за исключением:

- `sessionStorage` живёт до окончания `page session` (в разных браузерах этот термин трактуется по-разному, но в целом — пока браузер открыт);
- `localStorage` переживает закрытие браузера (данные не исчезают).

СОХРАНЕНИЕ ПО МЕРЕ НАБОРА

Что если мы хотим сохранять данные по мере набора?


Об этом поговорим чуть позже, при разборе событий `input` / `change`.

ПАРА СЛОВ ОБ HTML5

HTML5 настолько «стремительно» врывается в мир браузеров, принося с собой новые типы полей ввода, что браузеры до сих пор не договорились толком о том:

- поддерживать ли их;
- если поддерживать, то как;
- какие предоставлять свойства для стилизации.

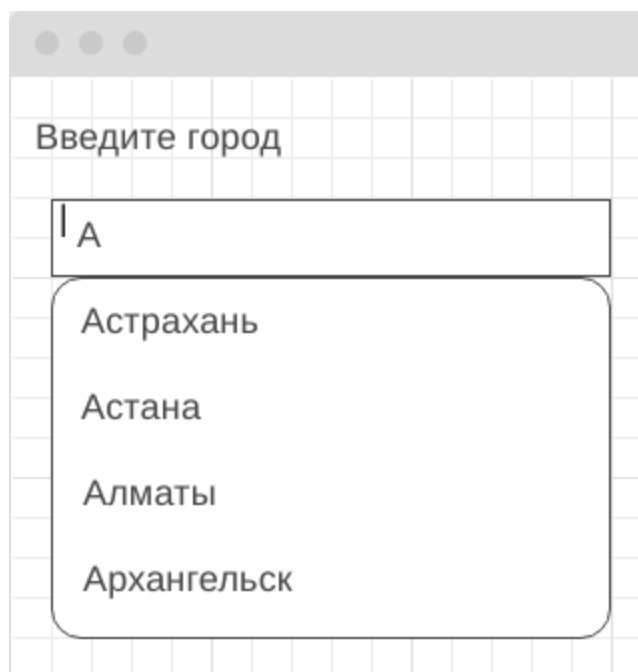
Поэтому в большинстве случаев вы будете использовать аналоги тех механизмов, которые мы рассмотрели для показа ошибок, для отображения всплывающих календарей, выпадающих списков и других интерактивных элементов.



Input vs Change

ЗАДАЧА

Делаем новую задачу: выбор города для какого-нибудь туристического портала. Реализовать форму нужно так, чтобы она выглядела следующим образом:



The image shows a UI mockup of a city selection form. It features a text input field with the placeholder text "Введите город" (Enter city). Below the input field, a dropdown menu is open, displaying a list of cities: Астрахань, Астана, Алматы, and Архангельск. The input field contains the letter "А".

Понятное дело, что фильтровать выводимый список нужно по мере набора, для этого нужно реагировать на соответствующие события.



ДОПУЩЕНИЕ

Представим, что массив городов у нас уже имеется (т.е. не нужно получать его с сервера), нужно лишь фильтровать список и отображать его на экране.

input VS change

Для реакции на ввод пользователя у нас есть событие `input`. Важная вещь: оно содержит только тип изменения и не содержит значение поля ввода в текущий момент времени. Получать его мы должны через `value`.

`change` возникает тогда, когда пользователь переключает фокус с поля ввода, поэтому с ним нужно быть достаточно аккуратным до отправки формы (для последнего поля).

ПОЗИЦИОНИРОВАНИЕ СПИСКА

Задача позиционирования списка решается так же, как и отображения ошибки:

- либо целиком через CSS + создание элемента добавление его в DOM через JS;
- либо позиционирование через JS.



ИТОГИ

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

- валидацию форм;
- подходы к отображению ошибок и кастомных элементов;
- кэширование данных в `Storage`.



Задавайте вопросы и напишите отзыв о лекции!

АЛЕКСАНДР ШЛЕЙКО

 a.shleyko@yandex.ru

 vk.com/shleiko

 [@dustyo_0](https://t.me/@dustyo_0)