FORMS

entering user data

Pedro Miguel Moreira

FORMS

entering user data

FORMS

Goals

- Understand the structure and elements of HTML forms.
- Learn DOM scripting for form processing with JavaScript.
- Style forms using CSS.
- Apply basic and advanced validation techniques.

FORMS Structure

FORMS Structure

- Basic function of a form:
  - `<form>`, `action`, and `method`.
- Example:

```
<form action="/submit" method="POST">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" required>
  <button type="submit">Submit</button>
</form>
```


FORMS Structure

- Basic organization of a form:
  - `<fieldset>`, `<legend>` and `<label>`
- Example:

```
<form action="/submit" method="POST">
  <fieldset>
  <legend>your opinion</legend>
      <label for="name">Name:</label>
      <input type="text" id="name" name="name" required>
  <fieldset>
  <legend>Choose your interests</legend>
  <div>
    <input type="checkbox" id="coding" name="interest" value="coding" checked />
    <label for="coding">Coding</label>
  </div>
  <div>
    <input type="checkbox" id="music" name="interest" value="music" />
    <label for="music">Music</label>
  </div>
</fieldset>
<button type="submit">Submit</button>
</form>
```


FORMS Structure

`<label>` Element

The `<label>` element provides a **text description for a form input**.

This description helps users understand what information they are supposed to enter in each field. Labels improve:

- **Clarity**: Labels tell users what each field is for, reducing confusion.
- **Accessibility**: Screen readers read out labels to help visually impaired users understand each field's purpose.

```
1 <label for="username">Username:</label>
2 <input type="text" id="username" name="username" required>
```

- To connect a label with a form input, the label's `for` attribute should match the input's `id`. This allows users to click on the label to focus on the input field, which is especially helpful for users with mobility issues.

FORMS Structure

`<fieldset>` Element and `<legend>` Element

The `<fieldset>` element is used to group several controls as well as labels within a web form.

The `<legend>` element represents a caption for the content of its parent `<fieldset>`element

```
  <fieldset>
    <legend>Choose your favorite monster</legend>

    <input type="radio" id="kraken" name="monster" value="K" />
    <label for="kraken">Kraken</label><br />

    <input type="radio" id="sasquatch" name="monster" value="S" />
    <label for="sasquatch">Sasquatch</label><br />

    <input type="radio" id="mothman" name="monster" value="M" />
    <label for="mothman">Mothman</label>
  </fieldset>
</form>
```


FORMS Input Elements

FORM Input Elements

`<input>` Element

Textual Inputs: General-purpose data entries (text, password, email).

Numeric Inputs: Specific for numbers and ranges.

Selection Inputs: Multiple (checkbox) or single (radio) selections.

Date/Time Inputs: Date, time, and combined date-time entries.

Specialized Inputs: Files and colors for specific tasks.

FORM Input Elements

`<input>` Element - Textual Inputs

`type="text"`

- **Purpose**: General-purpose text input.
- **Use Case**: Collecting short, single-line text entries such as names, usernames, and city names.

```html
<label for="username">Username:</label>
<input type="text" id="username" name="username">
```

- **list** attribute can be defined for input suggestions defined at a **data-list** element

javascript

```javascript
let username = document.getElementById('username').value;
```


FORM Input Elements

`<input>` Element - Textual Inputs

`type="password"`

- **Purpose**: Text input that hides the characters entered for privacy.
- **Use Case**: Password entry fields where text should not be visible.

```
1 <label for="password">Password:</label>
2 <input type="password" id="password" name="password">
```

- a **pattern** attribute can be defined for validation pruposes
- **minlenght** and **maxlenght** attributes can be defined

FORM Input Elements

`<input>` Element - Textual Inputs

`type="email"`

- **Purpose**: Specialized text input for email addresses, with built-in validation.
- **Use Case**: Ensuring users enter a valid email format (`user@example.com`).

```
1
2 <input type="email" id="email" name="email" required>
```

  - a **multiple** attribute can be defined to input a list of email addresses

FORM Input Elements

`<input>` Element - Textual Inputs

`type="url"`

- **Purpose**: Input for URLs, with built-in validation for proper format (e.g., `https://example.com`).
- **Use Case**: Collecting website links or personal URLs from users.

```
1 <label for="website">Website:</label>
2 <input type="url" id="website" name="website">
```


FORM Input Elements

`<input>` Element - Numeric Inputs

`type="number"`

- **Purpose**: Input for numeric values, allowing restrictions like `min` and `max`.
- **Use Case**: Collecting quantities, ages, and other numerical values.

```
1 <label for="age">Age:</label>
2 <input type="number" id="age" name="age" min="0" max="100">
```

- a **step** attribute can be defined to set the granularity (e.g. 2 in 2 numbers, or 0.1 in 0.1).
- Default step is 1, so by default, only integers are allowed.

FORM Input Elements

`<input>` Element - Range Inputs

`type="range"`

- **Purpose**: Slider input that allows users to pick a number within a range visually.
- **Use Case**: Settings that need a range of values (e.g., brightness, volume).

```html
<label for="volume">Volume:</label>
<input type="range" id="volume" name="volume" min="0" max="10">
```

- a **step** attribute can be defined

FORM Input Elements

`<input>` Element - Selection Inputs

`type="checkbox"`

- **Purpose**: Allows users to select multiple options independently (grouped by name attribute)
- **Use Case**: Multi-select options like interests, preferences, or consent agreements.

```
<label><input type="checkbox" name="interests" value="sports"> Sports</label>
<label><input type="checkbox" name="interests" value="music"> Music</label>
```


FORM Input Elements

`<input>` Element - Selection Inputs

`type="radio"`

- **Purpose**: Allows users to select only one option from a group (grouped by name attribute)
- **Use Case**: Single-choice options like gender, payment type, or survey responses.

```
<label><input type="radio" name="gender" value="male"> Male</label>
<label><input type="radio" name="gender" value="female"> Female</label>
<label><input type="radio" name="gender" value="other"> Other</label>
```


FORM Input Elements

`<input>` Element - Date and Time Inputs

`type="date"`

- **Purpose**: Calendar-based input for selecting dates.
- **Use Case**: Birth dates, appointment dates, scheduling, etc..

```html
<label for="dob">Date of Birth:</label>
<input type="date" id="dob" name="dob">
```

javascript

```javascript
const dateofbirth = document.getElementById('dob');
inputdate = dateofbirth.value       // get
dateofbirth.value = "2017-06-01"; // set
console.log(dateofbirth.value); // prints "2017-06-01"
console.log(dateofbirth.valueAsNumber); // prints 1496275200000, a JavaScript timestamp (ms)
```


FORM Input Elements

`<input>` Element - Date and Time Inputs

`type="time"`

- **Purpose**: Input for selecting time in hours and minutes.
- **Use Case**: Time selection for bookings, alarms, or events.

```
1 <label for="meeting-time">Meeting Time:</label>
2 <input type="time" id="meeting-time" name="meeting-time">
```


FORM Input Elements

`<input>` Element - Date and Time Inputs

`type="datetime-local"`

- **Purpose**: Allows selection of both date and time in a single field.
- **Use Case**: Scheduling events with specific dates and times.

```
1 <label for="appointment">Appointment:</label>
2 <input type="datetime-local" id="appointment" name="appointment">
```


FORM Input Elements

`<input>` Element - Specialized Inputs

`type="file"`

- **Purpose**: Allows users to upload files from their device.
- **Use Case**: Profile photo uploads, document submissions, or file attachments.

```
1 <label for="cv">Upload cv:</label>
2 <input type="file" id="cv" name="cv">
```


FORM Input Elements

`<input>` Element - Specialized Inputs

`type="color"`

- **Purpose**: Color picker for selecting colors visually.
- **Use Case**: Customizing user preferences like themes or UI colors.

```
1 <label for="favcolor">Favorite Color:</label>
2 <input type="color" id="favcolor" name="favcolor">
```


FORM Input Elements

`<input>` Element - Specialized Inputs

`type="hidden"`

- **Purpose**: send values to the server not visually presented to the user
- **Use Case**: item ids @db, not useful to the user, useful for the db / api calls.

```
<form>
  <div>
    <label for="title">Post title:</label>
    <input type="text" id="title" name="title" value="My excellent blog post" />
  </div>
  <div>
    <label for="content">Post content:</label>
    <textarea id="content" name="content" cols="60" rows="5">
This is the content of my excellent blog post. I hope you enjoy it!
    </textarea>
  </div>
  <div>
    <button type="submit">Update post</button>
  </div>
  <input type="hidden" id="postId" name="postId" value="34657" />
</form>
```


FORM `<textarea>` Element

`<textarea>` Multi-line Text Entry

Purpose : The `<textarea>` element is a **multi-line text input** field, unlike the single-line `<input type="text">`. It's ideal for situations where users need to enter a paragraph or longer text, such as comments, feedback, or descriptions.

Use Cases:

- comments or feedback, descriptions, messages, etc.

```
1 <label for="message">Message:</label>
2 <textarea id="message" name="message" rows="4" cols="50" placeholder="Enter your message here.
```

Attributes `rows` and `cols` set the height and width of the textarea

FORM <select> Element

<select> Dropdown Menus

Purpose : The <select> element creates a **dropdown menu** that allows users to choose from a list of predefined options. It's ideal for fields where the user must select from a set number of choices, which helps standardize data entry and prevent typos.

Use Cases:

- **Country selection ; Categories Language preference.**

```html
<label for="country">Country:</label>
<select id="country" name="country" required>
  <option value="us">United States</option>
  <option value="ca">Canada</option>
  <option value="uk">United Kingdom</option>
  <option value="au">Australia</option>
</select>
```


FORM <select> Element

<select> Dropdown Menus

- options can be grouped usin <optgroup>
- <hr /> can be used as a separator

```html
<label for="hr-select">Your favorite food</label> <br />
<select name="foods" id="hr-select">
  <option value="">Choose a food</option>
  <hr />
  <optgroup label="Fruit">
    <option value="apple">Apples</option>
    <option value="banana">Bananas</option>
    <option value="cherry">Cherries</option>
    <option value="damson">Damsons</option>
  </optgroup>
  <hr />
  <optgroup label="Meat">
    <option value="beef">Beef</option>
    <option value="chicken">Chicken</option>
    <option value="pork">Pork</option>
  </optgroup>
</select>
```


FORM submission

FORM submission

Form `action` and `method` attribute

- The `action` attribute specifies the **URL or endpoint** where form data should be sent once the form is submitted.
- It defines the destination of the form submission and typically points to:
  - a **server-side script** (e.g., in PHP, Node.js, Python))
  - *API endpoint* that will handle the data.
- the **URL** can be ***absolute*** or ***relative***

```
1 <form action="/submit" method="POST">
2 </form>
```

- the `method`attribute specifies **how** the form data will be sent to the server (the HTTP method) : **GET** or **POST**

FORM submission

GET vs POST

- `GET`: Sends data as URL parameters (query string). Suitable for non-sensitive data and when the form does not alter the server state.
- `POST`: Sends data in the request body, making it more secure and suitable for sensitive or large amounts of data.
  - Often used for form submissions that modify server state (e.g., creating a new user).

FORM submission

GET vs POST

GET:

- **Data is visible** in the URL (e.g., `https://example.com/submit?username=John`).
- Limited to a smaller amount of data due to URL length restrictions.
- **Use Case**: Search forms or filters where data visibility and bookmarking is useful.

```
<form action="/search" method="GET">
  <input type="text" name="query" placeholder="Search...">
  <button type="submit">Search</button>
</form>
```

POST:

- **Data is hidden** in the request body, not visible in the URL.
- Allows large amounts of data, such as form submissions with multiple fields.
- **Use Case**: Registration forms, login forms, or any data submission involving sensitive information.

```
<form action="/register" method="POST">
  <input type="text" name="username" placeholder="Username">
  <input type="password" name="password" placeholder="Password">
  <button type="submit">Register</button>
</form>
```


FORM submission

Best Practices

- **Always use `POST` for sensitive data** like passwords, emails, and personal details to ensure data is not visible in the URL.
- **Use descriptive URLs** in the `action` attribute to make form destinations clear and consistent.
- **CSRF Tokens**: When using `POST` for actions that modify data, consider implementing Cross-Site Request Forgery (CSRF) protection, as forms can be vulnerable to attacks.
- **Form Redirection**: On successful submission, redirect users to another page (e.g., confirmation or thank you page) to avoid resubmission on page refresh.

FORM submission

Form `enctype` attribute

`enctype` attribute defines how the data is encoded to be sent to the server

If the value of the `method` attribute is `post`, `enctype` is the MIME type of the form submission. Possible values:

- `application/x-www-form-urlencoded`: The default value.
- `multipart/form-data`: if the form contais files to upload
- `text/plain`: Useful for debugging purposes.

FORM submission

`<button>` element

A button element that can have three default behaviors according to the `type` attribute value

- `submit`: The button submits the form data to the server. This is the default.
- `reset`: The button resets all the controls to their initial values.
- `button`: The button has no default behavior.
  - Used to trigger further script processing (e.g submit the form using async requests, validate the form, etc.)

A button inside a `<form>` element is automatically associated to it. If located elsewhere association can be made using the `form` attribute which value must match the `name` attribute value of the `<form>` element

FORM Layout and Styling

Form Styling

- DigitalOcean How To Style Common Form Elements with CSS
- MDN Styling Web Forms
- MDN Advanced form styling

== exemplos de bibliotecas minimalistas ==

https://nielsvoogt.github.io/nice-forms.css/

https://picocss.com/

FORM Layout and Styling

best practices

- **keep it short**
- **layout** as a **single column** (usually the best option)
- **visually group related fields;** present it in a **logical sequence**.
- style **valid** and **invalid** inputs `:valid` `:invalid`
- provide **meaningful inline error messages**
- **use placeholders for hints** `::placeholder`
- **distinguish** between **required** vs **optional** fields
- **highlight the focus element** (`:focus`)
- Provide **autofill** and **autocorrect**.
- **Exclude all unneeded info** (text, pictorial, etc)
- avoid **reset** and **clear** buttons

Form Validation

Client Side FORM validation

Goal:

- ensure thar all (required) fields are filled in correct formats

Client-side validation

- is an initial check and an important feature of good user experience;
- by catching invalid data on the client-side, the user can fix it straight away, avoiding get data rejection from server
- should not be considered the only/last/ultimate secutity measure (usually is not difficult to bypass)

Client Side FORM validation

Typical messages from temptative form submission with invalid fields

- “This field is required” (You can’t leave this field blank).
- “Please enter your phone number in the format xxx-xxxx” (A specific data format is required for it to be considered valid).
- “Please enter a valid email address” (the data you entered is not in the right format).
- “Your password needs to be between 8 and 30 characters long and contain one uppercase letter, one symbol, and a number.” (A very specific data format is required for your data).

Client side validation focus on formats

Server side validation can look also at the coherence / correctness of the content

In some circumstances, real time server side supported validations (e.g using info @db), can be integrated via assynchronous requests.

Client Side FORM validation

Overview:

3 different kinds of client-side validation can be identified.

- **Built in HTML5 form validation** (built-in)
  - automatic, based on specific field types, and in attributes specifying formats and the compulsory nature of the form fields, usually triggered at submission, can be customized with javascript using the Constraints Validation API
- **Custom JavaScript global form validation**
  - implemented using javascript, enabling customization of more complex rules, usually triggered at submission, may also incorporate rules from the built-in HTML5 validation.
- **Custom JavaScript real-time (or live) form validation**
  - implemented using javascript, for each field or interrelated set of fields, triggered at the `input` or at the `blur` event. can make fetch requests to validate server side pieces of information before submission (e.g. existence of username)

Recommended : use robust HTML5 built-in form validation enhanced with javascript form validation, whenever needed or useful.

Note

There are several javascript form validation libraries available.

But, for now, you will learn how to do it by yourself.

Client Side FORM validation

Built in HTML5 form validation

- **When:** Automatically, as the form is submitted
- **How:** HTML5 form attributes can define which form controls are required and which format the user-entered data must be in to be valid.
- **Why:** HTML5 form validation is easy to implement without extra JavaScript, ensuring users to enter valid data types.
- **Examples:** an email field with proper format and required for submission.

Client Side FORM validation

Custom JavaScript global form validation

- **When**: Typically triggered during the form submission process.
- **How**: Using JavaScript to capture events like `submit`, and applying custom logic to validate each field based on specific rules
- **Why**: Enables more complex rules that HTML5 attributes can't handle, such as verifying password strength or comparing two fields for matching values, or enforcing a specific number or range of elements checked.
- **Example**: check if two passwords match.

Client Side FORM validation

Custom Javascript real-time (or live) Validation

- **When**: As the user is typing (often on the `input` event) or when they leave a field (on `blur`).
- **How**: JavaScript can provide immediate feedback by validating fields in real-time, showing success/error indicators, or dynamically updating messages.
- **Why**: Real-time feedback helps users correct errors on the go, reducing the chance of mistakes on submission and improving the overall form experience.
- **Example**: Displaying a warning if a username is too short while the user is still typing.

Client Side FORM validation

Built in HTML5 form validation

- `required`: Specifies whether a form field needs to be filled in before the form can be submitted.
- `minlength` and `maxlength`: Specifies the minimum and maximum length of textual data (strings).
- `min`, `max`, and `step`: Specifies the minimum and maximum values of numerical input types, and the increment, or step, for values, starting from the minimum.
- `type`: Specifies whether the data needs to be a number, an email address, or some other specific preset type.
- `pattern`: Specifies a regular expression that defines a pattern the entered data needs to follow.

Client Side FORM validation

Built in HTML5 form validation

When elements are valid

- The element matches the `:valid` CSS pseudo-class, which lets you apply a specific style to valid elements.
- The control will also match `:user-valid` if the user has interacted with the control,
- If the user tries to send the data, the browser will submit the form, provided there is nothing else stopping it from doing so (e.g., JavaScript).

Client Side FORM validation

Built in HTML5 form validation

When at least one elements is invalid

- Invalid elements match the `:invalid` CSS pseudo-class.
- If the user has interacted with a control, it also matches the `:user-invalid` CSS pseudo-class.
- If the user tries to send the data, the browser will block the form submission and display an error message.

Client Side FORM validation

Validating patterns (just some simple examples)

```
# Matches one character that is a (not b, not aa, and so on).
  a

# Matches a, followed by b, followed by c.
  abc

# Matches one char from the set {a,b,c}
  [abc]

# Matches a, optionally followed by a single b, followed by c. (ac or abc)
  ab?c

# Matches a, optionally followed by any number of b s, followed by c. (ac, abc, abbbbbc, and
  ab*c

# Matches exactly abc or exactly xyz (but not `abcxyz` or `a` or `y`, and so on).
  abc|xyz

# Matches 1 to 15 chars in (lower, upper, digit, underscore)
  [A-Za-z0-9_]{1,15}
```

> more info about **Regular Expressions** that you can test at **regexr.com**

Client Side FORM validation

Validating patterns (just some simple examples)

- matches banana, Banana, cherry or Cherry

CSS

```css
input:invalid {
  border: 2px dashed red;
}

input:valid {
  border: 2px solid black;
}
```

HTML

```html
<form>
  <label for="choose">Would you prefer a banana or a cherry?</label>
  <input id="choose" name="i-like" required pattern="[Bb]anana|[Cc]herry" />
  <button>Submit</button>
</form>
```


Client Side FORM validation

Validating patterns (example of more complex patterns)

```
# email username@{subdomain.}domain.iso
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$

# url http, https or ftp
^(https?|ftp):\/\/[^\s/$.?#].[^\s]*$

# numeric integer
^-?\d+$

# positive integ
^\d+$

# min 8 char with at least 1 lowercase , 1 uppercase, 1 number, and 1 special char,
^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[^\w\s]).{8,}$
```

> more info about **Regular Expressions** that you can test at **regexr.com**

Client Side FORM validation

Example of Built in HTML5 form validation

example of buil-in html5 form validation:

- **username** : required, must be 6-10 letters, no spaces or special characters.
- **email** : required must be a valid email address (e.g., example@example.com).
- **password** : required, must be at least 8 characters, include at least one uppercase letter, one lowercase letter, one number, and one special character.user interests
- **interests** : not required, select 0+
- **discount code**: not required exactly 6 chars (any char)

Explore **here a working example**

Client Side FORM validation

Example of custom javascript form validation

- **username** : required, must be 6-10 letters, no spaces or special characters.
- **email** : required must be a valid email address (e.g., example@example.com).
- **password** : required, must be at least 8 characters, include at least one uppercase letter, one lowercase letter, one number, and one special character.
- **password confirmation**: must be identical to password
- **interests** : not required, select +
- **discount code**: not required exactly 6 chars (any char)

Client Side FORM validation

example of custom javascript form validation workflow

```
// intercept the submission
document.getElementById('myForm').addEventListener('submit', function(event) {

  event.preventDefault(); // Prevent form submission
  let formIsValid = true; // keep a flag about its validity

  // proceed by validating field by field (group by group)

  const myfield = document.getElementById('myfield');            // the form field
  const myfieldError = document.getElementById('myfieldError');  // element to display errors

  // validation logic : this is a regular expression but could be anyother

  const myfieldErrorMsg = "Must be something"
  const myfieldPattern = /^[A-Za-z]{6,10}$/;
  if (!myfieldPattern.test(myfield.value)) {    // if it fails
    myfieldError.innerHTML = myfieldErrorMsg;    // show error message
    formIsValid = false;                         // update form validity
  } else {
    usernameError.innerHTML = '';                // else, clear message
  }

  /* ... validate other form fields  ... */

  // If all validations pass, allow form submission

  if (formIsValid) {
    form.submit();
  }
 });
```

> Explore **here a working example**

Client Side FORM validation

example of custom live javascript form validation workflow

```
// intercept with the input event
// display errors as user enters data
const dicountErrorMsg = "Must have 6 chars length"

document.getElementById('discount').addEventListener('input', function() {
  const discount = document.getElementById('discount');
  const discountError = document.getElementById('discountError');
  if (discount.value && discount.value.length !== 6) {
    discountError.innerHTML = dicountErrorMsg;
  } else {
    discountError.innerHTML = '';
  }
});

// intercept the form submission
document.getElementById('myForm').addEventListener('submit', function(event) {
  /** field by field validation **/
});
```

> Explore **here a working example**

Client Side FORM validation

XTRA JavaScript validation with Constraint Validation API

The Constraint Validation API consists of a set of methods and properties available on the following form element DOM interfaces:

The Constraint Validation API - availability

- `HTMLButtonElement` (represents a `` element)
- `HTMLFieldSetElement` (represents a `` element)
- `HTMLInputElement` (represents an `` element)
- `HTMLOutputElement` (represents an `` element)
- `HTMLSelectElement` (represents a `` element)
- `HTMLTextAreaElement` (represents a `` element)

Client Side FORM validation

XTRA JavaScript validation with Constraint Validation API

The Constraint Validation API - properties

- `validationMessage`: returns a localized message describing the constraints not satisfied (if any). (empty string if all constraints satisfied (is valid) or if it is not meant to be validated (`willvalidate` is false))
- `willValidate`: Returns `true` if the element will be validated when the form is submitted; `false` otherwise.
- `validity`:
  - returns a `ValidityState` object that contains several properties describing the validity state of the element, some examples follow:
    - `patternMismatch`: Returns `true` if the value does not match the specified `pattern`, `false` otherwise
    - `tooLong`/`tooShort`: Returns `true` if the value is longer/shorter than the value of `maxlength` /`minlength`, `false` otherwise.
    - `rangeOverflow` /`rangeUnderflow`:: Returns `true` if the value is greater / less than the value of `max` / `min` attribute, `false` otherwise.
    - `typeMismatch`: Returns `true` if does not fit the required syntax (when `type` is `email` or `url`), `false` otherwise.
    - `valid`: Returns `true` if it meets all validation constraints, and is therefore considered to be valid, `false` otherwise
    - `valueMissing`: Returns `true` if the element has a `required` attribute, but no value, `false` otherwise.

Client Side FORM validation

XTRA JavaScript validation with Constraint Validation API

The Constraint Validation API - methods

methods available on the above elements and the `form` element.

- `checkValidity()`: Returns `true` if the element's value has no validity problems; `false` otherwise. If invalid, this method also fires an `invalid` event on the element.
- `reportValidity()`: Reports invalid field(s) using events. This method is useful in combination with `preventDefault()` in an `onSubmit` event handler.
- `setCustomValidity(message)`: Adds a custom error message to the element;

Client Side FORM validation

more resources

> learn more: **Master JavaScript Form Validation: Enhance User Experience and Data Accuracy**

> learn more: **Mastering Form Validation with the Constraint Validation API**