Why we had to create a parent div ‘box’ and enclose form and ‘facts’ in it we could have apply flex properties on section element and could have two child divs one for form other for facts ?

Why we did not made to separate divs one for form other for facts, isntead of inclosing in box div?

why we did not enclosed form in a div ?

We didn't enclose the <form> in a <div> because the form itself is already a block-level element that can structure and contain input elements properly. Using a <div> would be unnecessary unless we needed an extra wrapper for styling

What is purpose of type attribute on input element and button , we could have used input elements solely why we had to inclose them in form element ?

On <input> Element:

The type attribute defines what kind of data the input field accepts.

On <button> Element:

The type of a button defines its behavior inside a form.

Using a <form> element provides several advantages:

When the form is submitted (via a button click or Enter keypress), the browser automatically sends the data to a server or handles it in JavaScript.

Without <form>, you’d need extra JavaScript to handle user input.

Pressing "Enter" inside an input field inside a form automatically triggers submission, making it more user-friendly.

Forms help group related input fields together. This is useful when handling multiple fields.

Are input elements inline elements , as button came right after input element?

Was it necessary to mention the width of section element ?

No, it wasn’t strictly necessary to mention the width of the <section> element because block-level elements (like <section>) naturally take up the full width of their parent container by default.

However, there are some possible reasons why width: 100% might have been explicitly added:

If the <section> is inside another container with a limited width, setting width: 100% ensures it stretches to its full available width.

Sometimes, developers add width: 100% for clarity, ensuring that future modifications don’t accidentally change the layout.

Why we had to apply width100% for form element when it’s width is already 100% ?

You're right that block elements (like <form>) naturally take up the full width of their parent container. However, explicitly setting width: 100% ensures consistency and prevents unexpected behavior.

Some CSS frameworks or browser default styles might modify the form's width.

Applying width: 100% ensures consistency across different browsers.

If the form is inside a flexbox or grid container, it may shrink based on content size.

Setting width: 100% ensures it always stretches to the intended width.

Why did we used outline: none property on input element , when it has apparently no effect , it looks the same even when this property is not applied there ?

Even though outline: none; does not visibly change the input field by default, it is used to remove the focus outline that appears when a user clicks or tabs into the input field.

By Default: Browsers add an outline (a blue or orange glow) when an input field is focused.

outline: none; removes this focus outline.

This is done mainly for design consistency, as the default browser outline may not match the website's theme.

Can we use padding-left: 1rem on input element , is there some reason in case of padding and padding left , rem is not used even though explicitly mentioning html font size ?

Yes, we can use padding-left: 1rem on the input element. There is no restriction on using rem for padding.

However, in most cases, px is preferred for padding because:

Padding is a spacing property, and designers often prefer absolute values (px) for better control.

rem is relative to the root html font size, so if the user changes the browser's default font size, the padding might scale unexpectedly.

px ensures consistency, while rem makes padding responsive, which may not always be desired.

Why did we targeted placeholder like this input[type="number"]::placeholder rather than input[type="number"] placeholder ?

::placeholder is a CSS pseudo-element specifically designed to target the placeholder text inside input fields.

🔹 placeholder (without ::) is not a child element inside <input>, so it cannot be selected like input placeholder {}.

We can not style placeholder {styling} like this

placeholder {

color: gray;

font-size: 16px;

}

This is invalid CSS ❌ because placeholder is not an HTML element—it is just an attribute of the <input> element.

it is necessary to mention input element because ::placeholder is a pseudo-element that must be attached to a valid HTML element.

Even though we set the font size for the html element , we are not utilizing it by using rem unit anywhere in file , if it was not needed why did we override the default font size of html element which is 16px ?

As we’re only using px, then this line is unnecessary and can be removed.

If we plan to use rem later, keeping it helps with readability and responsiveness.

Why did we applied display: inline-block; on button , and not on input element even though it is also form element ?

<button> → Default display is inline-block (but we explicitly set it to ensure it behaves as expected).

<input> → Default display is inline-block already, so we don't need to set it manually.

This property allows button to sit next to the <input> field instead of moving to the next line.

Without inline-block, if the button were accidentally treated as block, it would break onto a new line.

While most browsers treat <button> as inline-block by default, there are some cases where styles might be overridden:

Some CSS frameworks or resets (like Normalize.css) may force buttons to be block for accessibility reasons.

By explicitly setting inline-block, we ensure the button stays the way we intended (inline with the input)

What is effect of height: auto on facts class ?

It allows the height of the element to adjust dynamically based on its content. This means:

The .facts container will expand or shrink depending on the amount of text it contains.

Prevents Overflow Issues: Since the height is not fixed, it prevents content from being cut off or overflowing.

Why we didn’t used width:auto for .facts as well ?

We didn’t apply width: auto; to .facts because it's not necessary in this case

As block elements (like <div class="facts">) automatically take up the full width of their parent container unless specified otherwise.

Since .facts is inside .box, which already has width: 90%; max-width: 450px;, applying width: auto; wouldn't change anything.

width: auto; means "take up as much width as needed based on the content and available space."

But for block elements, this is already the default behavior unless width is explicitly set.

What If We Set width: auto;?

It wouldn't make a difference because .facts already expands to fill the entire width of its parent (.box).

Why we used e.preventDefault() ? is it property of event object only in case of when event type is submit ?

e.preventDefault() prevents the default behavior of an event.

In case of form submission event , the default behavior is:

The page reloads when you submit the form.

The browser sends the form data to the server (if an action is specified).

e.preventDefault() stops the default behavior of event (e.g in our case form submission reloading the page).

Without it, the page reloads, erasing user input in the form {number} before JavaScript can process it.

Preventing reload ensures that user-entered values remain accessible for API requests.

It allows JavaScript to fetch and display facts for the number which user entered in the form without interruption.

we can use e.preventDefault() with other events too.

For example in case of (<a> tags)

document.querySelector("a").addEventListener("click", function (e) {

e.preventDefault(); // Prevents navigation

console.log("Link clicked, but no navigation!");

});

Without e.preventDefault() → Clicking the link will navigate to the URL. Which is anchor tag’s default behaviour

With e.preventDefault() → Stops navigation, allowing custom behavior.

What is exactly happening in given statement const nummber = e.target.querySelector('input[type="number"]').value ?

e.target refers to the element that triggered the event.

In this case, since the event listener is attached to a <form>, e.target is the form element.

.value retrieves the current value inside the input field.

If the user types 42 in the number input, number will be "42" (as a string).

Why we used e.target.querySelector() rather than form.querySelector()

If we use form.querySelector(...), it always selects the first form on the page.

So even if we submit form2, it will still take values from form1 because form.querySelector(...) isn't checking which form was actually submitted.

if we have multiple forms, form.querySelector(...) always selects the first form.

But we have only one form on our page it will work fine

In case of e.target if we submit form2, e.target will correctly refer to form2 instead of always using form1.

e.target always refers to the exact form that was submitted, no matter how many forms exist on the page.

So if form2 is submitted, e.target is form2.

If form5 is submitted, e.target is form5.

How to add emogis in html and js file?

What did we add number in fetch(baseURL + number) ?

We are appending number to baseURL because the API requires a number in the URL to return facts about that number.

If we omit it api may return an error or a default response.

Given code would have given error ? const baseURL = `http://numbersapi.com/${number}`

No , we can also write it in template literals rather than using concatination operator +

Code will still work fine

Explain the following code

fetch('http://numbersapi.com/42')

.then(obj => obj.text())

.then(text => factDiv.innerHTML = text) each term

When we make a request to http://numbersapi.com/, the API responds with a plain text fact (not JSON).

When we call .text() on a Response object, it reads the body of the response and converts it into a plain text string.

.text() returns a Promise that resolves with the response body as a string.

The text contains the fact returned by the API.

We insert it into factDiv to display on the page.

Since the Numbers API returns plain text, not JSON, the actual raw response looks like this:

42 is the answer to the ultimate question of life, the universe, and everything.

When we use .text(), the Promise resolves with:

"42 is the answer to the ultimate question of life, the universe, and everything."

And we directly set this text in factDiv.innerHTML to display it on the page.

How do .then work even though there is lot of space between fetch’s closing parentheses ?

The .then() method works regardless of spaces or new lines

Unless there is a semicolon at the end of fetch or there is another statement in between fetch and .then , it breaks the chain in between them

Bcs is this case .then won’t be chained to fetch anymore

And can not receive promise

If we add semicolon at then end of 1st .then , evenn inn that case 2nd .then won’t be able to receive promise

Why did we made following changes in the code ?

const baseURL = 'https://cors-anywhere.herokuapp.com/http://numbersapi.com/'

fetch(baseURL + number , {

method:"GET" ,

headers:{

"x-requested-with" : "text/plain"

}

})

Explain each term in above code ?

***fetch(baseURL + number)***

So actual url is value in baseURL and nummber input from user

for example number = 5 then

fetch(https://cors-anywhere.herokuapp.com/http://numbersapi.com/5)

Kind of requesting facts for number 5 from api

***fetch(`https://cors-anywhere.herokuapp.com/http://numbersapi.com/${number}`***

We can also write fetch using template literals, which removes need of using string concatenation operator (+)

**BUT in following scenarios we should prefer other way {without template literals}**

As template literals were introduced in ES6 (2015) – Olde JavaScript environment {*it means Older Web Browsers , Outdated JavaScript Engines , JavaScript Running on Older Hardware*

} (pre-2015) does not support them

If the base URL is used multiple times, separating it as a baseURL variable can improve maintainability.

method:"GET"

By default, fetch() already uses GET, so this line is not necessary,and fetch will work completely fine, so we can omit it

But Some developers prefer writing it explicitly for readability.

If the project has other POST, PUT, or DELETE requests, keeping GET written explicitly makes all requests look uniform. So it is up to us

why did we added slash after 1st URL even though 1st URL ends with .com and why did we added slash at the end of second URL , why we enclosed whole URL in single quotes ?

**slash (/) between**

https://cors-anywhere.herokuapp.com/ and http://numbersapi.com/

Separates them correctly to avoid forming an invalid URL

**slash (/) afte**r numbersapi.com/ ensures that the number input is treated as a parameter rather than part of the domain name.

If we omit these in both cases URL’s will be treated as invalid domains, and the request will fail.

in JavaScript, **URLs are treated as strings** when passed to the fetch function, this is why enclosed with in quotes

Why we used this URL <https://cors-anywhere.herokuapp.com> before actual URL <http://numbersapi.com>

https://cors-anywhere.herokuapp.com

This is URL of a proxy server that helps us to bypass CORS (Cross-Origin Resource Sharing) restrictions.

CORS Anywhere forwards the request to the target API.

When the API responds, CORS Anywhere adds CORS headers in it

The modified response is sent back to your JavaScript, and the browser allows it.

**CORS restriction:** Many APIs block requests from different origins (domains) due to security policies.Browsers block cross-origin requests for security reasons (to prevent malicious websites from stealing data).

**Proxy server role:** This proxy forwards your request to the actual API and then sends the response back, making it look like the request came from the same origin.

Only omit the proxy if the API allows cross-origin requests or if you're running the request on a server-side environment (like Node.js) instead of the browser.

***step-by-step breakdown of how the CORS proxy server works***

Given fetch request Instead of making a request directly to <http://numbersapi.com/42>

The request first goes to https://cors-anywhere.herokuapp.com/.

The Proxy Server Receives Your Request

The proxy server (cors-anywhere.herokuapp.com) acts as a middleman.

It takes your request and forwards it to the actual API (http://numbersapi.com/42).

The API Receives the Request (From Proxy, Not Browser)

Since the API sees that a server (not a browser) is making the request,

It does not block the request (because CORS restrictions only apply to browsers).

The API processes the request and sends back the response to the proxy.

The Proxy Server Sends the Response Back to us

The proxy forwards the response it received from the API back to your JavaScript code.

Now your JavaScript can access the data without being blocked by CORS.

**Servers don’t have this CORS restriction, so the API doesn’t block the proxy server.**

**Same-origin requests (allowed)A request is considered same-origin when it comes from the same protocol, domain, and port.**

In following scenario A user visits https://mywebsite.com/page (a webpage on mywebsite.com).

That webpage makes a request to https://mywebsite.com/api/data (an API on the same domain).

Request from Request to

https://mywebsite.com/page → https://mywebsite.com/api/data

https://mywebsite.com/page → /api/data

Cross-origin requests (blocked unless allowed by CORS):

https://mywebsite.com → https://anotherwebsite.com

https://mywebsite.com → https://api.mywebsite.com

"x-requested-with" : "text/plain"

It is often used when making requests through a CORS proxy server

Some proxy servers (like cors-anywhere.herokuapp.com) require a special header to allow requests.

"x-requested-with": "text/plain" is a custom header that tricks the proxy into thinking the request is coming from a trusted source.

Without it, the proxy might reject the request.

If the API already supports CORS, we don’t need a proxy or this header.

This is only necessary because the API doesn’t allow cross-origin requests.

Many servers and proxies use "x-requested-with" as a way to identify AJAX (fetch/XHR) requests.

Originally, browsers like jQuery and XMLHttpRequest automatically sent this header with AJAX requests.

Now, many proxies expect this header for security reasons.

Many proxy servers check for specific headers to decide whether to allow a request. One common check is for the "x-requested-with" header.

Some APIs or proxies assume that requests with this header come from a trusted source (like a real browser or an approved app).

If the header is missing, the proxy might block the request.

The value "text/plain" doesn't actually affect how the request is processed—it’s just a trick to bypass restrictions on some proxy servers.

What is CORS?

CORS (Cross-Origin Resource Sharing) is a security feature built into web browsers which enables browsers to restrict web pages from making requests to an other website which has different domain.

Browsers enforces CORS that prevents a website (yourwebsite.com) from making requests to another website (api.com) unless api.com(server) explicitly allows it.

How It Works Step-by-Step:

You open your browser and visit https://yourwebsite.com.

Your JavaScript code tries to fetch data from https://api.com/data.

Your browser checks if api.com allows cross-origin requests.

If api.com does NOT send the CORS headers in it’s response , your browser blocks the request.

The browser does NOT block the request from being sent to the API—the request successfully reaches the server.

The API receives the request and may even send back a response.

But if the response does not include the correct CORS headers,

The browser blocks your JavaScript from accessing the response.

Browsers block responses from APIs that don’t include CORS headers.

Browsers enforce CORS (Cross-Origin Resource Sharing) to prevent malicious websites from stealing data from other sites.

The API (https://some-api.com) processes your request and sends a response

Before your JavaScript can access the response, the browser checks for CORS headers in the response.

If this header is missing, the browser sees that the API does not allow cross-origin requests.

Why I got error undefined in following part of code ?

fetch(baseURL)

.then(response=> response.json())

.then(obj=> factDiv.innerHTML = obj.contents)

.catch(error => factDiv.innerHTML = "Error fetching facts")

The fact is inside obj.contents, not obj.data.

There is no such property in returned JavaScript object which is called data

In case of this api requested information is in property content

Make sure to revise promises topic from joke

Explain the given code with full depth , each line and term

const baseURL = 'https://api.allorigins.win/get?url=' + encodeURIComponent(`http://numbersapi.com/${number}`);

fetch(baseURL)

.then(response=> response.json())

.then(obj=> factDiv.innerHTML = obj.contents)

.catch(error => factDiv.innerHTML = "Error fetching facts")

})

***'https://api.allorigins.win/get?url=' + encodeURIComponent(`http://numbersapi.com/${number}`)***

we didn't write this code exactly like the previous one because AllOrigins API works differently from CORS Anywhere proxy.

The CORS Anywhere proxy expects a full URL in the path (after the /).

The proxy forwards the request exactly as it is, acting as a direct middleman.

AllOrigins API doesn’t accept full URLs in the path

AllOrigins is built to accept URLs as a query parameter(?url=...).

Since the API URL (http://numbersapi.com/42) needs to be inside a query string (?url=...), we must encode special characters (:, /), or else they might get misinterpreted.

***https://api.allorigins.win/get*** → This is the proxy server's URL.

***?url=*** → This is a query parameter that tells the proxy which actual API you want to fetch data from.

***encodeURIComponent('http://numbersapi.com/${number}')*** → This ensures that the API URL is properly formatted and passed as a parameter.

A query parameter is a key-value pair added to a URL after a ? (question mark) to send additional information to the server.

APIs often need extra details (like user input) to return the correct data.

Example: <https://api.weather.com?city=London>

**https://api.weather.com** → This is the base URL of the API.

**?city=London** → This is a query parameter that tells the API you want weather data for London.

So, when you request this URL, the API will respond with only London's weather data instead of the weather for all cities.

If you wanted data for New York instead, you'd modify the query parameter

Some APIs (like AllOrigins) are designed to accept the target URL as a query parameter instead of appending it with /.

How allorigins Proxy Server Interacts with API (Step by Step)

fetch(baseURL) sends a request to AllOrigins proxy (https://api.allorigins.win/get).

Proxy Server Receives the Request

The proxy server (AllOrigins):Extracts the target API URL from the query parameter (?url=http://numbersapi.com/42).

And sends a request to the actual API (http://numbersapi.com/42)

API does NOT block it, because it is coming from a server (not a browser, which would be blocked by CORS).

API Server Processes the Request

The target API (http://numbersapi.com/42):

Sends the response back to the proxy server.

The API does not add CORS headers, because it thinks it is talking to another server.

Once the proxy server receives the response from the target API:

It modifies the response to include CORS headers (Access-Control-Allow-Origin: \*).

It packages the data in a JSON format.

Your browser receives the response from the proxy server:

The response now includes CORS headers, so the browser allows JavaScript to access it.

The response is in JSON format (converted from the raw API response).

Why we used encodeURIComponent()

encodeURIComponent() is required when an API URL is inside a query parameter, to prevent breaking the request.

If we do not use it , browser sees ?url=http://numbersapi.com/42.

But http:// has :, /, and ?, which can break the URL because : and ? have special meanings in URLs.

Some servers might think /42 is another parameter instead of part of the API URL.

Due to : server might think http: is a separate part of the query, not the start of a URL.

Without encoding: ❌ The proxy might misinterpret the API URL.

❌ It may truncate the URL, send a wrong request, or return an error.

With encoding: ✅ The API URL is treated as a single, correct parameter.

as encodeURIComponent() converts these special chractars

The browser and proxy correctly understand the API URL as a single parameter.

Why we did not used encoding in previous code

In the previous code, we didn't need encodeURIComponent() because we directly appended the API URL to the proxy using a slash (/) instead of passing it as a query parameter.

No query parameters (?url=...) were involved, so special characters (: and /) were not an issue.

In this case we are using query parameter

query parameters can misinterpret characters like :, /, and ?, we need encodeURIComponent() to ensure the URL is treated as a single string.

If the proxy lets you append the API URL with / → No encoding needed.

If the proxy expects the API URL as a query parameter (?url=) → Encoding is required to avoid breaking the URL.

Why Use +

Concatenating the base URL and the encoded API URL

'https://api.allorigins.win/get?url=' → The base proxy server URL

encodeURIComponent(\http://numbersapi.com/${number}\`)` → The encoded API URL

+ joins them into one full URL.

URLs are always passed as strings, no matter the function.

A URL is just text that represents an address.

Functions like fetch() and encodeURIComponent() expect a string.

It is a string , as fetch expects a stringas argument

'https://api.allorigins.win/get?url=' + encodeURIComponent(`http://numbersapi.com/${number}`)

**response.json():** Converts response object to JavaScript object

factDiv.innerHTML = **obj.contents** here contents is a property of JavaScript object , and this property’s value is actual fact returned from api

Why we omitted 2nd argument of fetch in this version of code ?

In the previous version, we used a proxy (cors-anywhere.herokuapp.com) that required the "x-requested-with" header.

With api.allorigins.win, we don’t need extra headers.

And we omit GET method as it was unnecessary

So apparently we do not need 2nd argument

Unlike CORS Anywhere, which sometimes requires extra headers (like "x-requested-with"), api.allorigins.win is specifically designed for cross-origin requests and automatically handles CORS.

Some proxies (like CORS Anywhere) block requests unless they detect a specific header, which is why "x-requested-with" was used before.

AllOrigins doesn't require this trick because it is built to allow any request.

Handles CORS Automatically

It modifies the API response to include Access-Control-Allow-Origin: \*, so the browser never blocks the response.