Why did we apply width and max-width two properties in container?

Applying both width and max-width to .container make layout more responsive and flexible.

**width: 90%** ensures responsiveness on small screens.

**max-width: 500px** prevents the container from getting too large on large screens.

If we only set max-width: 500px and do not set width: 90%, then:

**On large screens:**

The .container will be 500px wide, which is fine.

**On small screens:**

The .container will stay at 500px even if the screen width is smaller than 500px.

This can cause overflow issues where part of the container goes off-screen, requiring horizontal scrolling

**Why max-width: 500px?**

It prevents the container from getting too wide on large screens.

Even if 90% of the screen width is more than 500px, the max-width caps it at 500px.

This keeps the design readable and visually appealing on big screens.

How it is choose between both widths , that which one should be applied?

The choice between width: 90% and max-width: 500px depends on how CSS applies these properties together.

**CSS Rules for width and max-width Together**

The browser chooses the smaller of the two values.

The browser calculates

If 90% is less than 500px, then width: 90% is used.

If 90% is greater than 500px, then max-width: 500px is enforced instead.

**Case 1: Small Screen (320px width)**

90% of 320px → 288px

max-width is 500px

width: 90% is applied (because 288px < 500px)

**Case 2: Medium Screen (600px width)**

90% of 600px → 540px

max-width is 500px

max-width: 500px is applied (because 540px > 500px)

In case of other combinations

|  |  |
| --- | --- |
| **Combination** | **How It Works** |
| width + max-width | Chooses the smaller value |
| width + min-width | Chooses the larger value |
| min-width + max-width | Ensures width stays between the two values |
| Two width values | The last one declared wins |

Even though container is already in the center why did we applied margin 0 auto property?

It ensures centering when Flexbox is not available or changes.

If display: flex property is removed from parent element or if another parent container is added that affects alignment, margin: 0 auto; ensures the container still stays centered horizontally.

So even though Flexbox centers the .container, margin: 0 auto; is a backup mechanism that ensures horizontal centering remains intact even if layout changes. It's a best practice for robustness!

Why did’t we used min-height = 100vh in percentage unit too as we used for width =100%

**width: 100%** means the element takes up the full width of its parent (which is usually the <body> or <html> in this case).

**height: 100%** only works if all parent elements have a defined height.

<html> and <body> do not have a default height, so height: 100% on <section> may collapse.

all parent elements (like <html> and <body>) have a default width, but they do not have a default height (their height only grows based on the content inside).

as parent element doesn’t have a defined height, then height: 100% on a child element doesn’t know what 100% refers to. As a result, the browser defaults the child’s height to 0px—which means it collapses.

When we say an element "collapses", it means the element's height becomes 0px, making it invisible or taking up no space on the page.

If you want height: 100% to work, you must set a height on all parent elements:

html, body {

height: 100%; /\* Ensures parents have a height \*/

}

section {

height: 100%; /\* Now 100% means full viewport height \*/

}

Why we did not used width in vw unit and used it in percentage in section ?

|  |  |
| --- | --- |
| **Property** | **What It Does** |
| width: 100% | Takes 100% of the parent element's width. If the  parent is <body>, it covers the full available width  (without causing issues) |
| width: 100vw | Takes 100% of the viewport width, regardless of  the parent. This can sometimes cause  unintended scrolling. |

Why we did not used min-width instead of width in percentage in section ?

**1. Prevents Unwanted Horizontal Overflow**

**width: 100%** strictly limits the section to the full width of its parent (usually <body>).

**min-width: 100%** allows the section to expand beyond 100% if its content is wider, which can cause horizontal scrolling issues.

**2. Ensures Proper Layout Sizing**

**width: 100%** makes sure the section never exceeds the parent’s width, keeping everything contained.

**min-width**: 100% does not restrict maximum width, which means elements inside could make it expand beyond the screen size.

we could have used max-height or height why we choosd min-height instead in section element ?

|  |  |
| --- | --- |
| **min-height: 100vh** | Ensures the section is at least 100% of the  viewport height but can grow if content  overflows. |
| **max-height: 100vh** | Ensures the section is at most 100% of the  viewport height, meaning content might be cut  off if it overflows. |
| **height: 100vh** | The section must always be exactly 100% of the viewport height (even if the content inside is larger).it may cause overflow |

What is difference between margin= auto 0 and margin = 0 auto ?

**margin: 0 auto;**

A way to center an element horizontally.

0 → No margin on top and bottom

auto → Equal margin on left and right, which centers the element horizontally.

.container {

width: 50%;

margin: 0 auto;

background-color: lightblue;

}

The container takes 50% width.meaning there is 50% of the screen left as empty space.

auto distributes that empty space equally on both sides, centering the element

margin: 0 auto; does NOT work unless the element has a defined width (less than 100%).

By default, block-level elements (like <div>, <section>, etc.) take up 100% of the available width.

If an element is already 100% wide, there is no extra space on the left or right to distribute, so auto margins do nothing.

Setting width: 50% (or any value less than 100%) creates space for auto to distribute and center the element.

width does not have to be in percentage (%)! It can be in pixels (px), viewport width (vw), em, rem, etc.

**margin: auto 0;**

useful for centering an element vertically only when it has a defined height and it has a parent element which has display flex property

auto → Tries to apply equal margin on top and bottom, which only works if the element has a defined height.

0 → No margin on left and right.

Without display: flex; margin: auto 0; does NOT work as expected!

Use justify-content: center; or align-items: center; when centering ALL elements inside a container.

Use margin: auto 0; when you only want ONE element to be centered vertically without affecting others.

When we apply width = 90% in container why did it’s width did changed but it was kind of centered from both side instead of width increasing from one side and should not have any gap from both sides?

When you apply width: 90% to .container, its width changes relative to its parent element, but it stays centered instead of expanding only to one side because

The .container has margin: 0 auto;, which automatically distributes the remaining space equally on both sides.

Even though the .container is 90% of the parent’s width, the remaining 10% of the width is divided equally as margins on both sides (5% on each side).

What does it mean by lineheight = 3rem?

line-height: 3rem; means that the vertical space between each line of text in an element is 3 rem (relative to the root font size).

If the root font size is 10px, then 3rem = 3 × 10px = 30px of vertical space per line.

Why Use rem Instead of px?

rem is relative to the root element (<html>), while px is an absolute unit.

If the user increases the default font size in their browser for better readability, rem units will scale accordingly, but px values will not.

(By default, most browsers set the root font size (<html>) to 16px unless the user changes it in the settings.

When You Set html { font-size: 10px; }

You override the browser’s default root font size and force it to be 10px.

It defeats the purpose of using rem for accessibility!

The user's custom font-size preference gets ignored because you’re forcing 10px instead of letting the browser control it.

Best Practice

Instead of hardcoding px in html, use percentages so it scales properly:

html {

font-size: 62.5%; /\* 62.5% of 16px (default) = 10px \*/

}

)

If later you decide to adjust the base font size (html { font-size: 12px; }),

the rem values automatically adjust, but px values require manual changes.

What does it mean by text-align = justify ?

text-align: justify; makes the text align evenly along both the left and right margins by adjusting the spaces between words.

Normally, text aligns left by default.

With justify, the browser adds extra spaces between words so that both the left and right edges of the text line up evenly.

p {

text-align: left;

width: 300px;

}

output:Left aligned, but the right edge is uneven.

This is a simple example

of text alignment. The

right side is uneven.

p {

text-align: justify;

width: 300px;

}

Text output:

Left and right aligned, spaces between words adjusted.

This is a simple example

of text alignment. The

right side is now even.

What does it mean by width = fit-content, what are some other rare values for width like this ?

Why we using rem unit rather than px , is it a good practice which one should be preferred ?

why we are able to apply border radius even though we have literally removed the border at 1st place in the following scenario

border: none;

border-radius: 4px;

What does it mean by align-self: flex-end?

Why did we had to mention the class of parent div , can’t we specify button directly ? if we can is it a bad practice that is why it was’t done at 1st place in following scenario

const button = document.querySelector('.container button')

document.addEventListener('DOMContentLoaded' , getJoke)

What does it mean by DOMcontentloaad in ablove scenerio ?

DOMContentLoaded is an event that fires when the HTML document has been completely loaded and parsed, but before images, stylesheets, and other external resources are fully loaded.

In this case, it ensures that getJoke() runs as soon as the HTML is ready (instead of waiting for everything like images to load).

Without DOMContentLoaded, if the JavaScript runs before the HTML is fully loaded, it might try to access elements (.container p) that don’t exist yet, causing errors.

why did we omit parentheses from function call getJoke when passing as a an argument ?

Without parentheses (getJoke) → We pass a reference to the function. The function will be called automatically when the event occurs.

With parentheses (getJoke()) → It would execute immediately instead of waiting for the event.

We omit parentheses to pass the function itself, rather than calling it immediately.

why we did not closed the 2nd argument which is a function in single quotes as we did for 1st argument , can we use double quotes also at place of single quotes ?

The first argument ('DOMContentLoaded') is a string because it’s the name of an event.

The second argument (getJoke) is a function reference, not a string.

If we put the function name in quotes ('getJoke'), JavaScript would treat it as a string and not execute the function.

Both single (') and double (") quotes work the same way in JavaScript when defining strings.

function getJoke{

fetch('https://icanhazdadjoke.com/' ,{

headers:{

'accept' : 'application/json'

}

}).then(data=> data.json())

.then(obj=> jokeText.innerHTML = obj.joke)

}

Explain the above code with full details with each line and word with full depth

fetch('https://icanhazdadjoke.com/' ,{

headers:{

'accept' : 'application/json'

}

})

fetch('https://icanhazdadjoke.com/', {...})

fetch() is a built-in JavaScript function that sends an HTTP request to a server and returns a Promise.

* The first argument is the URL of the API
* The second argument is an object that can be method ,headers , body and many more

Are method, headers, body, etc., Properties or Objects?

In the second argument of fetch(), we pass an unnamed object (also called the "options object").

Inside this object:

method, body, mode, cache, redirect, credentials → are properties (key-value pairs).

headers is a property, but its value is another unnamed object.

In JavaScript, when we pass an object directly as an argument, we don’t need to give it a name.

The second argument of fetch() is an unnamed object.

It’s just { key: value, key: value } inside fetch().

A Promise is a JavaScript object that represents a value that will be available in the future.

When you make an HTTP request using fetch(), it does not return the actual data immediately. Instead, it returns a Promise, which acts like a placeholder for the data that will arrive later.which will later contain the response.

Since a Promise does not give the data instantly, we need to use .then() to handle it once it resolves.

.then() method waits for the Promise to resolve, then executes the function inside it.

If the Promise fails, .catch() will handle the error.

At first, the Promise is "pending" (it has no final value yet).

When a Promise resolves, it means that async operation completed successfully.

When a Promise resolves, it means:

The Promise now has an actual value instead of being a placeholder.

That value can be used in the next step of the code .then

For example, when fetch() successfully retrieves data from an API

Promise is resolved with a Response object

Response object is the value which is passed to promise placeholder

It contains metadata { status , type URL, headers ,body(actual data) }

Response {

type: "cors",

url: "https://icanhazdadjoke.com/",

status: 200,

ok: true,

headers: Headers {},

body: ReadableStream,

...

}

A JSON (JavaScript Object Notation) object is simply data written in a structured format that looks like JavaScript objects.

Example JSON object:

{

"id": "12345",

"joke": "Why don’t skeletons fight each other? They don’t have the guts!"

}

It consists of key-value pairs, like "id": "12345" and "joke": "..."

JSON is a text-based format, commonly used for exchanging data between a server and a web application.

When we request data from an API, it often returns JSON data as a response.

Promise Failed (Rejected)

If something goes wrong (e.g., no internet, API is down, wrong URL), the Promise fails (rejects), and .catch() is triggered.

HTTP request

An HTTP request is a message sent from a web browser or a script (like JavaScript) to a server to ask for data or perform an action.

When you visit a website, your browser sends an HTTP request to the website’s server, and the server responds with a webpage, data, or other resources.

In our case, we are using JavaScript’s fetch() function to send an HTTP request to an API server to get a joke.

There are different types of HTTP requests, but the most common ones are:

* GET → Fetches data from a server (✅ This is what we are using in the joke generator).
* POST → Sends new data to a server.
* PUT → Updates existing data on a server.
* DELETE → Removes data from a server.

In JavaScript, when using the fetch() function:

If no method is specified, the request is automatically a GET request.

GET is the default method for fetch().

To Explicitly Write GET?

Even though it's not necessary, we can explicitly specify the method like this:

fetch('https://icanhazdadjoke.com/', {

method: 'GET', // Explicitly saying "this is a GET request"

headers: {

'Accept': 'application/json'

}

})

How Would It Look for Other HTTP Methods?

If we were sending data using POST, we would have to explicitly write:

method: 'POST',

API (Application Programming Interface)

An API is a service that provides data when requested. The API URL is the address where browser send the request.

APIs return data in different formats, such as:

JSON (application/json) , XML (application/xml) , Plain Text (text/plain) , HTML (text/html)

The Dad Joke API (https://icanhazdadjoke.com/) by default, returns data in HTML format.

But if we send a request with the Accept: application/json header, it returns JSON

headers: { 'accept': 'application/json' }

Headers (object) are additional information sent with an HTTP request.

They tell the server how we want to send or receive data.

By default, if you visit 'https://icanhazdadjoke.com/' in a browser, it returns an HTML page.

We don’t want HTML, we want JSON (structured data).

The header tells the server:

"Hey, I want the response in JSON format!"

'accept' → The name of the header

It’s lowercase because HTTP headers are case-insensitive.

'application/json' → The value of the header.

It tells the server we expect a JSON response instead of plain text or HTML.

Some common headers include :

Accept Specifies the type of response format (JSON, XML, HTML, etc.).

Content-Type Specifies the format of the data we are sending in the request body.

Authorization and many more

Even though there is only one headers keyword in the code, it contains multiple key-value pairs inside the object {}.

Because the headers object is a collection of key-value pairs, each representing a different header.

We only use one headers keyword, but inside it, we define multiple headers as key-value pairs.

Each key-value pair inside headers represents a different header.

fetch('https://example.com/api/data', {

method: 'POST',

headers: { // Single "headers" keyword, but multiple key-value pairs inside

'Content-Type': 'application/json',

'Accept': 'application/json',

'Authorization': 'Bearer YOUR\_API\_KEY',

'User-Agent': 'Mozilla/5.0'

},

body: JSON.stringify({ name: "John Doe", age: 30 })

})

In above example headers object contains four different headers:

'Content-Type': 'application/json'

'Accept': 'application/json'

'Authorization': 'Bearer YOUR\_API\_KEY'

'User-Agent': 'Mozilla/5.0'

The HTTP header names follow a specific format and are usually case-insensitive.

All HTTP header names and values should be enclosed in quotes to follow proper JSON formatting and avoid errors.

Without quotes, JavaScript might treat it as a variable name instead of a string.

Quotes are needed for property names in objects when they contain special characters(such as - in our case ) or follow a specific format (like HTTP headers).

Header values must always be strings.

Even numeric values should be converted to strings , this is why 'application/json'

is also quoted as it is header value and a string

.then(data=> data.json())

data.json() returns a Promise, which eventually resolves to the actual JSON object.

In a fat arrow function data=> data.json() , if the function body only contains a single expression, the return keyword is implicitly added. We don’t need to write it explicitly

In above code data=> data.json() is a fat arrow function passed as an argument

In JavaScript in case of arrow functions, if there's only one parameter, we can omit the parentheses.

As we did so for data parameter in above code

We can omit parentheses only when there’s one parameter.

If there are zero or multiple parameters, we must use parentheses

.json() function is a method on the Response object that:

Extracts the body (which is json format) from the Response object

Parses the JSON text into a JavaScript object

Since extraction and parsing takes time, JavaScript doesn't block execution. Instead, it returns a Promise that resolves when the process is complete.

In short . json is a asynchronous function

.json() does not touch the metadata (like status, headers).

In Response object only body is in json format

When you call data.json(), it returns a Promise that eventually resolves to a JavaScript object.

.then(obj=> jokeText.innerHTML = obj.joke)

obj.joke refers to the "joke" property in the object, which contains the joke text.

How Do We Know the Property Name?

Check the API Documentation

Every API has documentation explaining its response structure.

Example: icanhazdadjoke.com specifies that the joke is under the "joke" property.

Different APIs have different properties—always check their documentation or inspect the response.

async function getJoke(){

    const jokeData = await fetch('https://icanhazdadjoke.com/' ,{

        headers:{

            'Accept' : 'application/json'

        }

    });

    const jokeobj= await jokeData.json();

    jokeText.innerHTML = jokeobj.joke;

}

Explain above code

The keyword async is used to define an asynchronous function.

It allows the function to use await inside it.

Without async, await will cause a syntax error.

An async function always returns a Promise (implicitly, even if we don’t explicitly return one).

You can use async and .then() together, but it's unnecessary.

await makes JavaScript wait for the Promise to resolve before moving to the next line.

Without await, JavaScript would move to the next line before receiving a response.

fetch() starts fetching the data, but since we did not await it, jokeData holds a Promise, not the actual response.

console.log(jokeData) prints Promise { <pending> }, because fetch() is still in progress.

jokeData.json() is called, but since we didn’t await, it also returns a Promise, not actual data.

Why await before jokeData.json()?

Because json() also returns a Promise.

await makes JavaScript wait until the JSON data is fully converted into a JavaScript object.

Without await, JavaScript would move to the next line before the conversion is complete.