**FORM FACTOR**

**Some factors about form factor to be considered while developing,**

* **Different forms, different screens**

The most noticeable development is the physical one, that of the form factor. Developers have to deal with smart devices that come in different shapes and sizes, from the smallest wearable to the largest smart TV and everything in between. So, whether it’s a smartphone, phablet, tablet or laptop, developers have to consider the outcome of their work on different screen sizes. A website for instance should be responsive, so it easily adapts to the corresponding resolution. Then, it takes quite some effort to make sure that all visual elements are displayed correctly and of course also function likewise.

* **Fragmentation**

Next to the physical differences, there is the software. The operating systems that run the apps and software define in a large manner how your app will work. However, fragmentation is a huge challenge. In terms of OS-functionalities and capabilities, this is essential and determining for the development process.

* **Mobility**

The number of mobile internet users has increased immensely compared to a decade ago. Convenient as it is to be connected all the time, the speed with which we are connected differs, depending on a subset of factors. This ranges from the build of the device and the network service (4G/LTE) to the ISP and the locally corresponding Wi-Fi connectivity. It’s important for dev teams to take this into consideration when designing or building an app or a website. How does low connectivity affect performance? What are the load times and how can these be reduced efficiently? These are important questions to ask when creating software for mobile devices.

* **Functionalities**

There’s a myriad of smart devices out there. All have their own designs and features. Depending on the functional design of your app or website, developers can create new software with these features in mind. One could think of touch interfaces, NFC, GPS, Bluetooth, gyroscope, audio, video or lighting features for example. Of course, these features also depend on the type of device. A functional design for a stationary smart TV or desktop computer differs from one for mobile devices, such as smart wear, mobile phones or tablets. Not only does this affect the device itself, it also influences the way the device is used. What if the software is used on-the-go? Does the app need an online connection or can it use offline capabilities?

* **Dedicated copy**

Several things are important when creating copy for apps or websites. First, be sure who you’re writing your copy for. Accordingly, the tone of voice and overall imagery of your software should match. Second, as smartphones have smaller screens than laptops, make sure that you keep the text clear and compact when developing for smaller form factors. For mobile websites, text can be reduced and adjusted in a responsive design. Less is more applies very well in this case. Also be aware to use the right interaction terms for the right media. When browsing on a desktop computer, one should not read instructions for hand gestures, such as swiping and such. This may seem arbitrary, but good and corresponding copy in the end enhances the overall experience and retains users.

* **User-friendly navigation**

Lastly, it should be easy to use an app or website. When developing an alternative navigation, offer guidance in the form of a small tutorial or tooltips. In case you do have a lot of data to display, make sure to include helpful options, such as a search bar or filtering functionalities.

Something that also enhances the user experience is to match an app’s design with the OS it was built for. It’s important to take the design principles into account when developing an app. This way, it’s more intuitive for end-users to recognize familiar patterns and actions.

**GRID LAYOUT**

A grid is a collection of horizontal and vertical lines creating a pattern against which we can line up our design elements. They help us to create designs where elements don’t jump around or change width as we move from page to page, providing greater consistency on our websites.

A grid will typically have **columns**, **rows**, and then gaps between each row and column — commonly referred to as **gutters**.



**Defining a grid**

To define a grid we use the **grid** value of the [**display**](https://developer.mozilla.org/en-US/docs/Web/CSS/display) property. Unlike flexbox, the items will not immediately look any different.

.container {

display: grid;

}

Declaring display: grid gives you a one column grid, so your items will continue to display one below the other as they do in normal flow. To see something that looks more grid-like, we will need to add some columns to the grid.

.container {

display: grid;

grid-template-columns: 200px 200px 200px;

}

**Flexible grids with the fr unit**

.container {

display: grid;

grid-template-columns: 1fr 1fr 1fr;

}

**Gaps between tracks**

To create gaps between tracks we use the properties [**grid-column-gap**](https://developer.mozilla.org/en-US/docs/Web/CSS/grid-column-gap) for gaps between columns, [**grid-row-gap**](https://developer.mozilla.org/en-US/docs/Web/CSS/grid-row-gap) for gaps between rows, and [**grid-gap**](https://developer.mozilla.org/en-US/docs/Web/CSS/grid-gap) to set both at once.

.container {

display: grid;

grid-template-columns: 2fr 1fr 1fr;

grid-gap: 20px;

}

**Repeating track listings**

You can repeat all, or a section of, your track listing using **repeat** notation. Change your track listing to the following:

.container {

display: grid;

grid-template-columns: repeat (3, 1fr);

grid-gap: 20px;

}

You will now get 3 **1fr** tracks just as before. The first value passed to the repeat function is how many times you want the listing to repeat, while the second value is a track listing, which may be one or more tracks that you want to repeat.

**The implicit and explicit grid**

We have only specified column tracks so far, and yet rows are being created to hold our content. This is an example of the explicit versus the implicit grid. The explicit grid is the one that you create using grid-template-columns or grid-template-rows. The implicit grid is created when content is placed outside of that grid — such as into our rows. The explicit and implicit grids are analogous to the main and cross flexbox axes.

By default, tracks created in the implicit grid are auto sized, which in general means that they are large enough to fit their content. If you wish to give implicit grid tracks a size you can use the [grid-auto-rows](https://developer.mozilla.org/en-US/docs/Web/CSS/grid-auto-rows) and [grid-auto-columns](https://developer.mozilla.org/en-US/docs/Web/CSS/grid-auto-columns) properties. If you add grid-auto-rows with a value of 100px to your CSS, you will see that those created rows are now 100 pixels tall.

.container {

display: grid;

grid-template-columns: repeat (3, 1fr);

grid-auto-rows: 100px;

grid-gap: 20px;

}

**The minmax () function**

Our 100-pixel tall tracks won’t be very useful if we add content into those tracks that is taller than 100 pixels, in which case it would cause an overflow. It might be better to have tracks that are at least 100 pixels tall and can still expand if more content gets into them. A fairly basic fact about the web is that you never really know how tall something is going to be; additional content or larger font sizes can cause problems with designs that attempt to be pixel perfect in every dimension.

The minmax function lets us set a minimum and maximum size for a track, for example minmax(100px, auto). The minimum size is 100 pixels, but the maximum is auto, which will expand to fit the content. Try changing grid-auto-rows to use a minmax value:

.container {

display: grid;

grid-template-columns: repeat(3, 1fr);

grid-auto-rows: minmax(100px, auto);

grid-gap: 20px;

}

If you add extra content you will see that the track expands to allow it to fit. Note that the expansion happens right along the row.

**As many columns as will fit**

We can combine some of the things we have learned about track listing, repeat notation and minmax() to create a useful pattern. Sometimes it is helpful to be able to ask grid to create as many columns as will fit into the container. We do this by setting the value of grid-template-columns using repeat() notation, but instead of passing in a number, pass in the keyword auto-fill. For the second parameter of the function we use minmax(), with a minimum value equal to the minimum track size that we would like to have, and a maximum of 1fr.

Try this in your file now, using the below CSS:

.container {

display: grid;

grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));

grid-auto-rows: minmax(100px, auto);

grid-gap: 20px;

}

**The HTML DOM (Document Object Model)**

When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page.

The HTMLDOM model is constructed as a tree of **Objects**:

### **The HTML DOM Tree of Objects**



## The CSS Box Model

All HTML elements can be considered as boxes. In CSS, the term "box model" is used when talking about design and layout.

The CSS box model is essentially a box that wraps around every HTML element. It consists of: margins, borders, padding, and the actual content. The image below illustrates the box model:

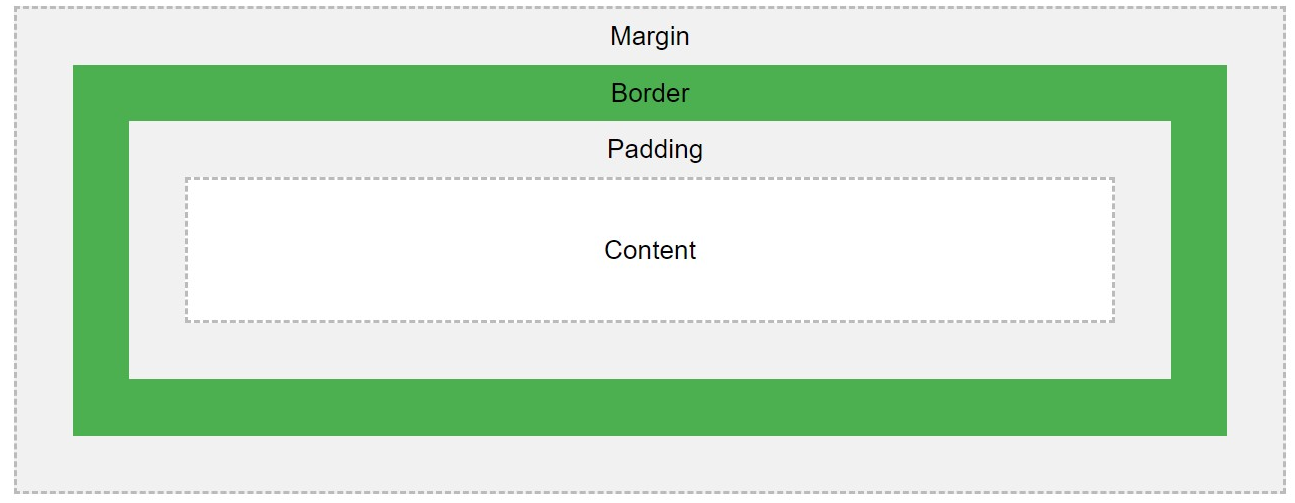
Explanation of the different parts:

* **Content** - The content of the box, where text and images appear
* **Padding** - Clears an area around the content. The padding is transparent
* **Border** - A border that goes around the padding and content
* **Margin** - Clears an area outside the border. The margin is transparent

The box model allows us to add a border around elements, and to define space between elements.

Example for box model:

div {  
  width: 300px;  
  border: 15px solid green;  
  padding: 50px;  
  margin: 20px;  
}

****

**Semantics**: It deals with the perception of the

audience with regards to the way the content is structured.

Some of the benefits from writing semantic markup are as follows:

* Search engines will consider its contents as important keywords to influence the page's search rankings
* Screen readers can use it as a signpost to help visually impaired users navigate a page
* Finding blocks of meaningful code is significantly easier than searching though endless divs with or without semantic or name spaced classes
* Suggests to the developer the type of data that will be populated
* Semantic naming mirrors proper custom element/component naming

These are some of the roughly **100** semantic elements available:

* <article>
* [<aside>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/aside)
* [<details>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/details)
* [<figcaption>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/figcaption)
* [<figure>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/figure)
* [<footer>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/footer)
* [<header>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/header)
* [<main>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/main)
* [<mark>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/mark)
* [<nav>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/nav)
* [<section>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/section)
* [<summary>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/summary)
* [<time>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/time)



**Accessibility**: It means developing content to be as accessible as possible no matter an individual's physical and cognitive abilities and no matter how they access the web.

The Web is fundamentally designed to work for all people, whatever their hardware, software, language, culture, location, or physical or mental ability. When the Web meets this goal, it is accessible to people with a diverse range of hearing, movement, sight, and cognitive ability.

**Q) What happens at the backend when you enter on URL?**

A) The URL has mainly two parts the **protocol** and the domain name. The domain name is further divided into **TLD** (Top Level Domain) and Second Level Domain.

For example, https is the protocol, ‘.com’ is the TLD, and google is the second level domain.

Each and every device has an **IP** address i.e., Internet Protocol address. This address is used to request access to that particular system, whether it is a desktop, smartphone, laptop or server.

When an URL is hit in the browser, the browser doesn’t know the IP address of the website we are trying to access. So, it sends the URL to the **DNS** (Domain Name Service) servers via the **ISP** (Internet Service Providers). The DNS servers send back the IP address associated with the URL via the ISP.

The **browser** which now has the IP address accesses the required HTML file stored in the server specified by the IP address.

**Q) Which are the previous versions of HTML5 and why are we in HTML5?**

A) When did HTML5 come out? It’s not really a question of when HTML5 was released, since it’s more of a collection of evolutionary features than a single, revolutionary package. This means that HTML5 has been around in some form since as far back as 2008, even though its official specification wasn’t finalised until 2014.

Unlike previous HTML versions, HTML5 has been developed by two separate groups: the **World Wide Web Consortium** (**W3C**) and the **Web Hypertext Application Technology Working Group** (**WHATWG**).

To understand the advancements of HTML5, we need a brief overview of the HTML timeline:

**1991–1999**: HTML is created by web legend Tim Berners-Lee in 1991, and HTML versions 1–4 are developed throughout the 1990s by W3C. In these early days of widespread internet use, HTML efficiently displays the vast majority of web content, since at this time it largely consists of static, non-interactive sites.

**2000**: W3C recommends XHTML 1.0 – an XML-based markup language that mirrors/extends HTML. Previous versions of HTML are now showing their age, struggling to handle the latest generation of multimedia, interactive sites. To get the best results, developers are resorting to third-party plugins.

**2004**: Development of HTML is closed by W3C when it decides to instead focus on XHTML. WHATWG is formed to develop HTML further, with the aim of reflecting the modern dynamic web, while keeping backwards compatibility with existing HTML code.

**2004–2006**: WHATWG gains support from major web browser developers. In 2006, W3C also announces its support for the project.

**2008**: The first public draft of HTML5 is released by WHATWG.

**2012**: W3C and WHATWG decide on separate development of HTML5. W3C would work on a definitive standard of HTML5, while WHATWG would pursue development of a ‘living standard’ – a continual evolution with ongoing improvements.

**2014**: W3C releases its official [HTML5 recommendation](https://www.w3.org/TR/html5/).

**Q) Why we need HTML5?**

* HTML5 offers a host of new features including special elements for semantic labelling such as <section>, <nav>, <article>, <aside>, <header>, <footer> and <main> – these define the type of content associated with each element. This makes structuring content more efficient and helps search engines crawl the website, meaning a well-structured HTML5 site has better chances of ranking highly in search results.
* Adding multimedia elements is also simplified with <audio> and <video> tags, and a wide selection of new application program interfaces (APIs) lets developers easily integrate a huge range of functionality into HTML5 sites.
* One of the big advantages of HTML5 is that it doesn’t rely on proprietary third-party plugins to create complex and powerful web projects.