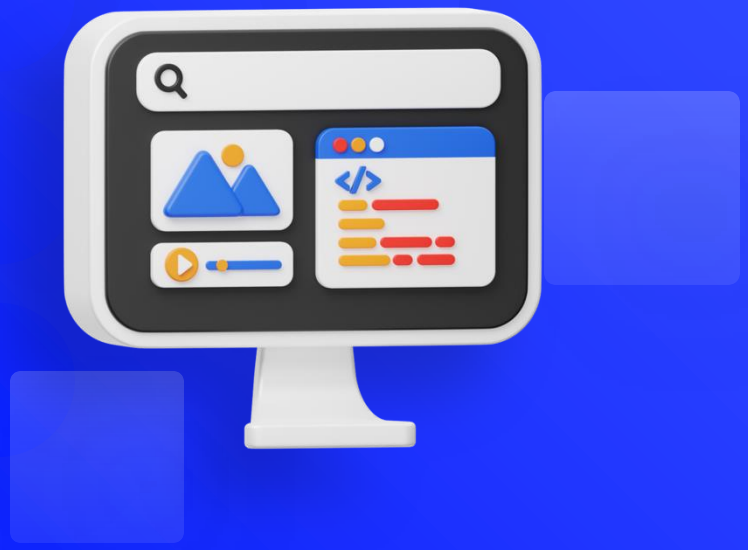
Institute of Data

Software Engineering

Module 4 More on Front-End Development

Introduction



As you start your journey as a web developer, you will be tempted to improvise and hope for the best. While this is a common practice for beginners, soon you will discover that planning is the most important thing you can do in this industry.

Whether you are working on your side hustle or a client project, on a complex back-end or an intuitive and feature-rich front-end, you should always have a good plan.

In web development, the plan starts with the **Design**.

Creating a good design requires having a thorough understanding of the project **requirements**. Often these two processes will feed into each other, to build up a complete and accurate picture of the application **before** any development is started.

Requirements Gathering

When building your application, try to find the best way for yourself to “make others” understand what it is you want to deliver.

Not one single path is the best and each project may look different, but here is one way of doing things.

The process should clarify a complete set of requirements, and capture how each function relates. It then feeds into design that illustrates how the app will look and work.

1. What are the key requirements ? For example, an e-shop requires that you have stock, a buy button, or a cart management system. List them all and describe the main parts.
2. Sketch the application flow on paper, trying to understand how everything communicates, and what causes something else to trigger. For example, buying a product needs to automatically update the stock list, or if the user can login, they also need to be able to sign up.
3. Define the data models and ensure your lo-fi designs include all required data elements.
4. Use any tool you prefer to create a prototype, and present it to someone who is not you to test. Testing your own work is a terrible practice.
5. Find ideas and define a colour -> colour scheme.
6. Create the HI-FI designs.

What is Design?

Design is many things. Following the dictionary: "**a plan or drawing produced to show the look and function or workings of a building, garment, or other object before it is made.**"

Then what do we as **software engineers** mean by design?

In the IT industry we have many types of designs, here we will focus only on a few:

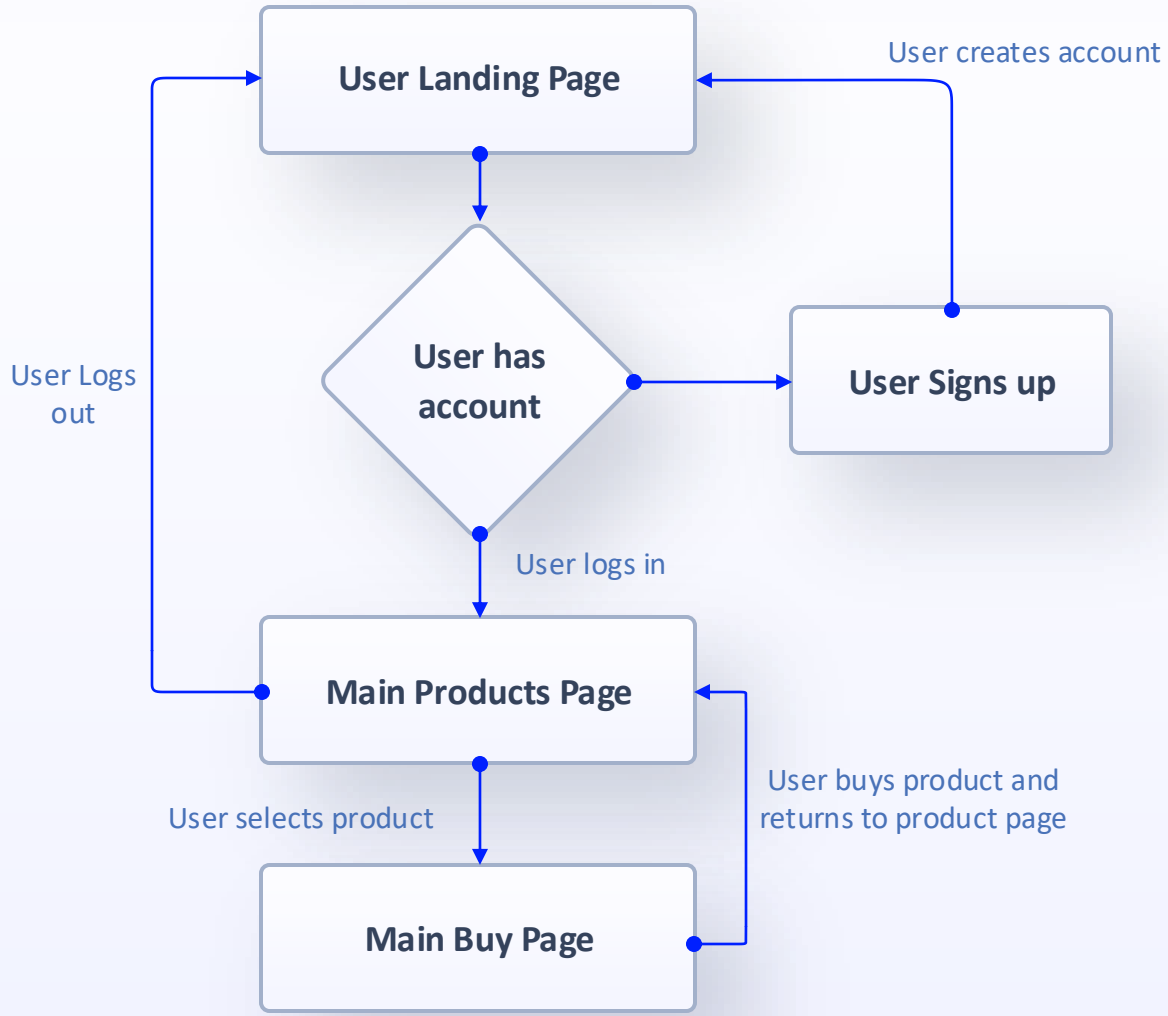
- Application flow design
- Sketches
- LoFi (Low Fidelity) designs
- HiFi (High Fidelity) designs
- Paper prototype design

Application Design

The application flow design is the most important part of your application. Here you can find out if your application will do what it is meant to do. Do you know how many developers **forget to create a sign up page**?

Having the correct structure will let you easily understand how much work there is to get done.

It can be hard not to jump in and start coding, but we are still very far away.



Application Design

A good **application flow diagram** needs to start with a good understanding of your user's **identity** and **goals**. It should put the user's perspective at the forefront of the design process and serve as a **common language** that everyone on the project can understand.

Visualising the user's path early helps identify and address issues, **saving time and resources** on later fixes. You will often need **multiple flow diagrams** for different features to keep things clear.

Rectangles, the most common shape in user flows, represent screens like homepages or confirmation pages, indicating no action is required.

Arrowed lines connect shapes and guide the reader through the chart, indicating flow from top to bottom or left to right.

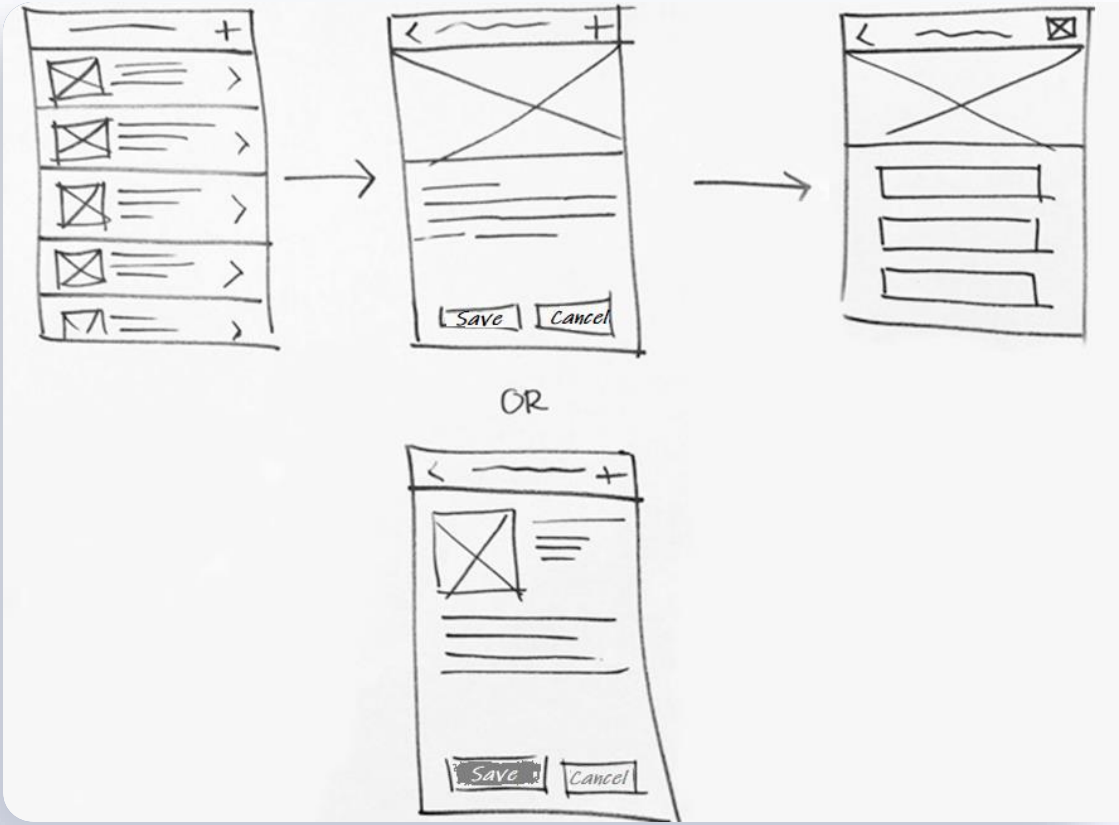
Circles represent actions, tasks, or processes that must be completed.

Diamonds, or "decision diamonds," represent questions that dictate the flow based on possible answers, such as "Yes" or "No."

Sketches

Sketches are quick, hand-drawn wireframe representations of the app's interface, focusing on layout and functionality rather than aesthetics. They allow designers to brainstorm and iterate ideas rapidly, laying the groundwork for more refined designs.

Any time you spend on your application must create value. If the value you create in 5 minutes is the same as 2 hours, then you should only spend 5 minutes. **What is value?** Anything that helps you move forward. In early stages, pretty pictures don’t do that.



Lo-Fi Designs

Lo-Fi is the key to acceptance, and also shows that you are professionally trained. Low Fidelity should be as Low as possible.

Hints - don't use colours, in fact, they should be ugly, as plain and ugly as possible, because you are showcasing functionalities.

When people see your lo-fi, you don't need to explain it is lo-fi. The moment you try to make lo-fi look attractive, you will move the attention from functionality to look and feel. **You don't want that**.

These are lo-fi designs created through Balsamiq. There are many applications you can use. We will explore FIGMA, which is the standard in the industry, but overall, you should find something you like and can easily work with.

Also remember, you are a **web developer**, not a designer, you need to know the process and the tools, but ultimately, it is not your job to make things look pretty.

Paper prototyping

Paper prototyping involves design teams creating paper versions of digital products to understand concepts and test designs. They draw sketches or adapt printed materials for low-fidelity samples to guide designs and study user reactions early on.

Even in the high-tech digital UX world, pen and paper remain favoured for quick low-fidelity prototyping. UX teams spend time away from computers, using sticky notes, whiteboards, and notepads to annotate paper prototypes.

Effective planning lets designers swiftly create wireframes, mockups, and prototypes after this preliminary phase.

Paper prototyping is crucial for early UX design as it encourages collaboration and cost-effective exploration of various ideas.

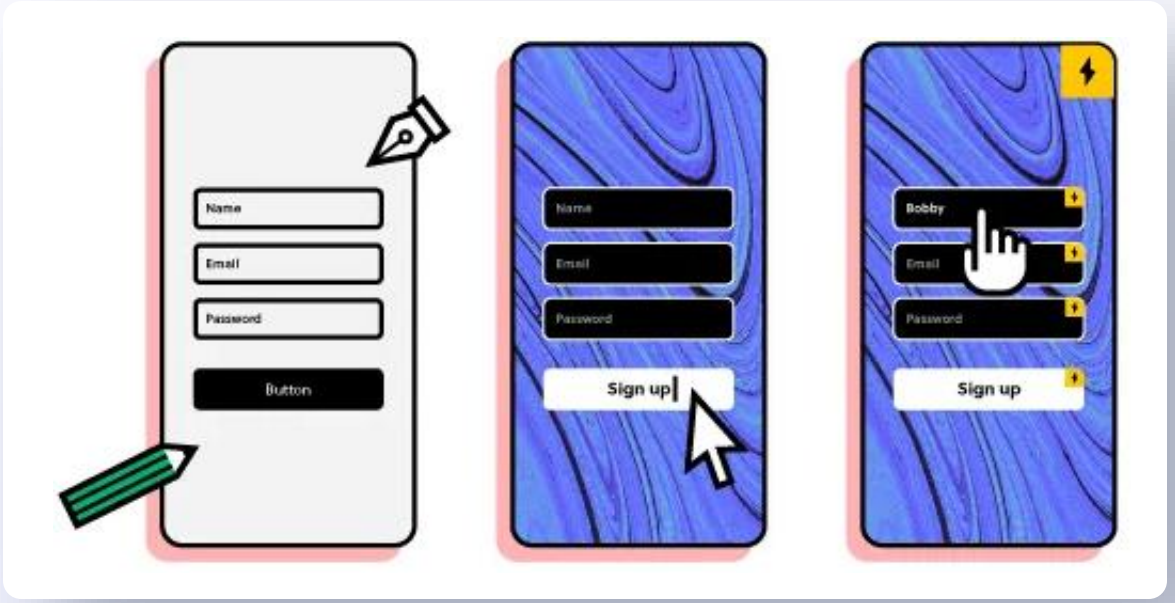
Hi-Fi Designs

High-fidelity designs are polished representations of the final product, complete with colours, typography, and interactive elements.

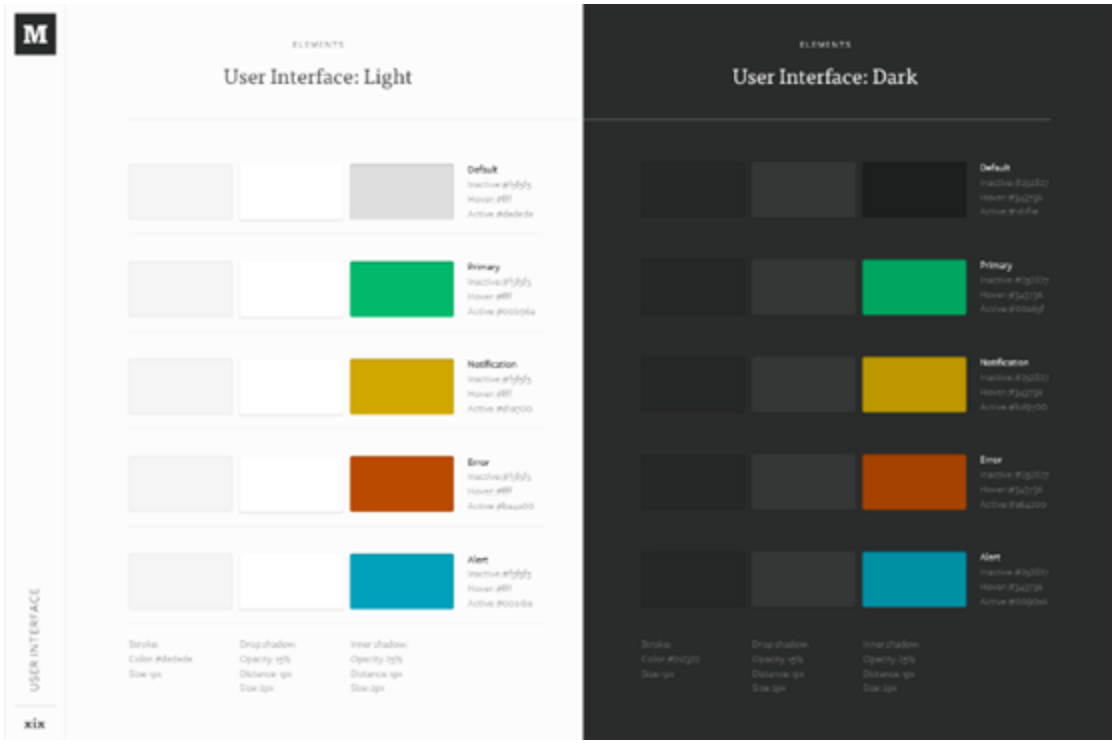
These designs closely mimic the end-user experience, focusing on useability, intuitiveness, and accessibility. They are often used for stakeholder presentations, user testing, and development handoff.

Although it will not often be your job as a web developer to create Hi-Fi designs, often you will need to work with them to implement the design into functionality, especially as a front-end developer.

Being familiar with common Hi-Fi design tools such as Figma, Canva, and the Adobe suite can increase your productivity.



Style Guides



Style Guides are sets of rules and standards that define the **visual** and **written** elements of a brand or project, ensuring **consistency** across designs, content, and communication.

They often include detailed specifications on how things like **logos**, **colours**, **fonts** and **images** should look to ensure consistent branding and identity representation.

Style Guides

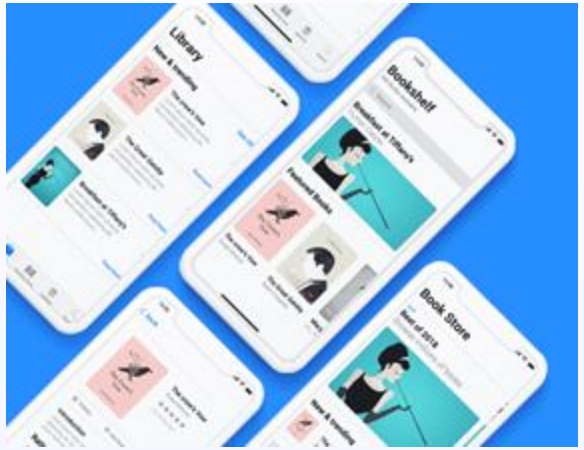
You may wonder why so many applications all look the same, particularly if they represent the same company. That is because we make use of **style guides**. Almost every major company has a well defined and DOCUMENTED style guide (see here for some examples).

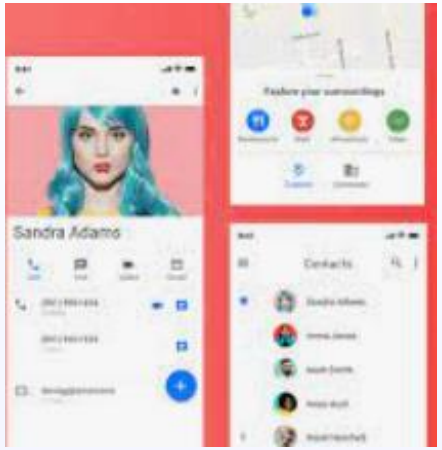
Although not precisely a style guide, in this module we will use “Bootstrap”, because it is a widely used, more generic styling framework. It provides the accessibility and consistency benefits of a style guide and it will help you to easily transition everywhere.

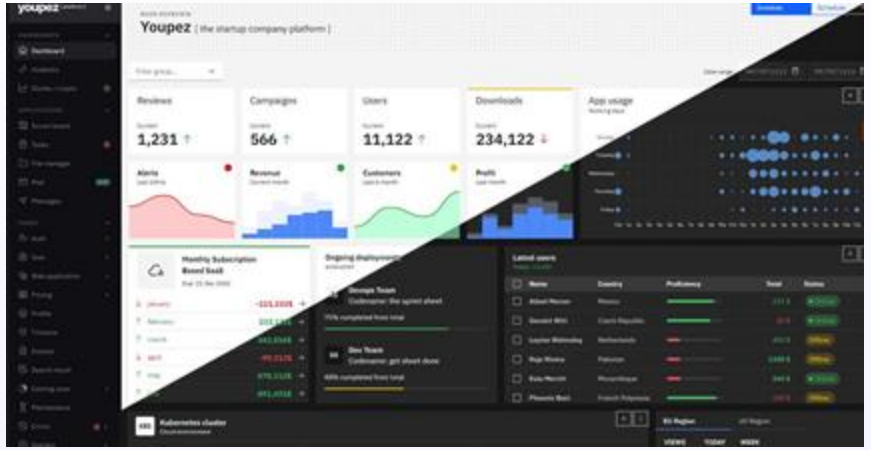
Do some research into current web design trends and share with the class a style that you really like. (Hint: Awwwards is a great source for modern web trends)

Class Exercise: Some examples

Major companies name their style guides. Google has **Material**, Microsoft has Fluent, IBM has **Carbon**, Apple has **HIG**. Can you tell which of those companies each screenshot belongs to?

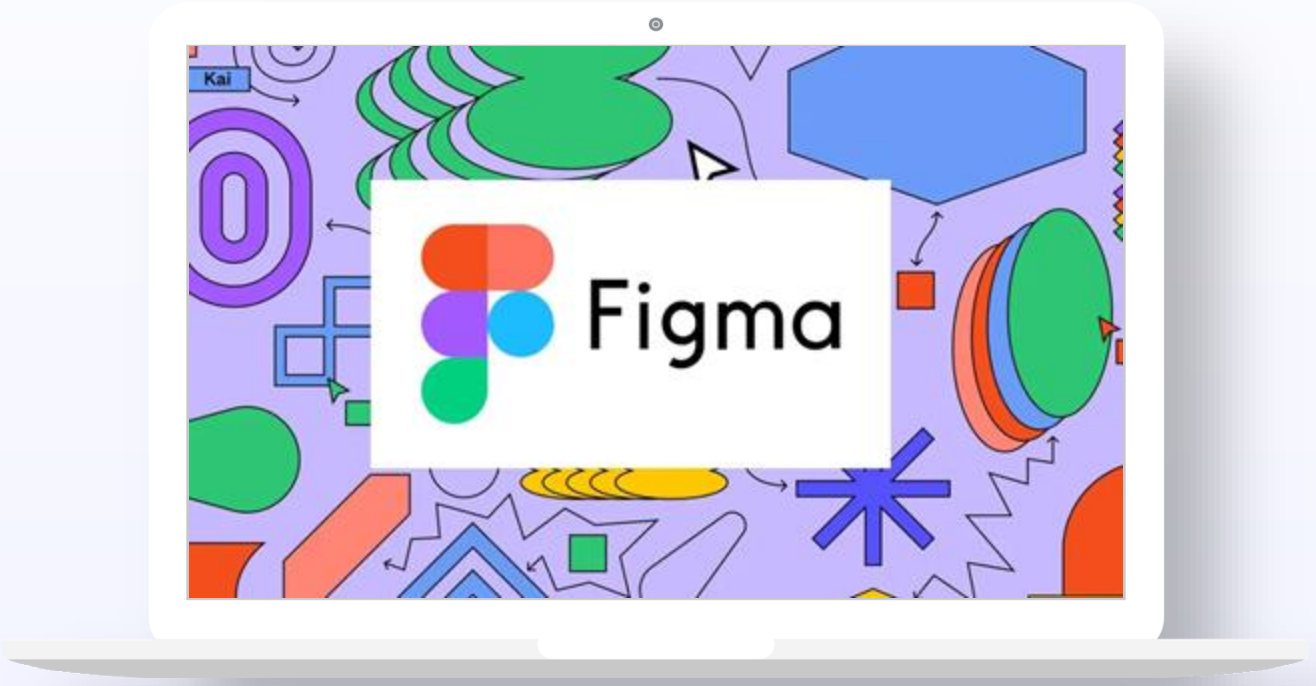






Figma

Figma is a popular industry tool to help both designers and developers brainstorm, design and build applications.



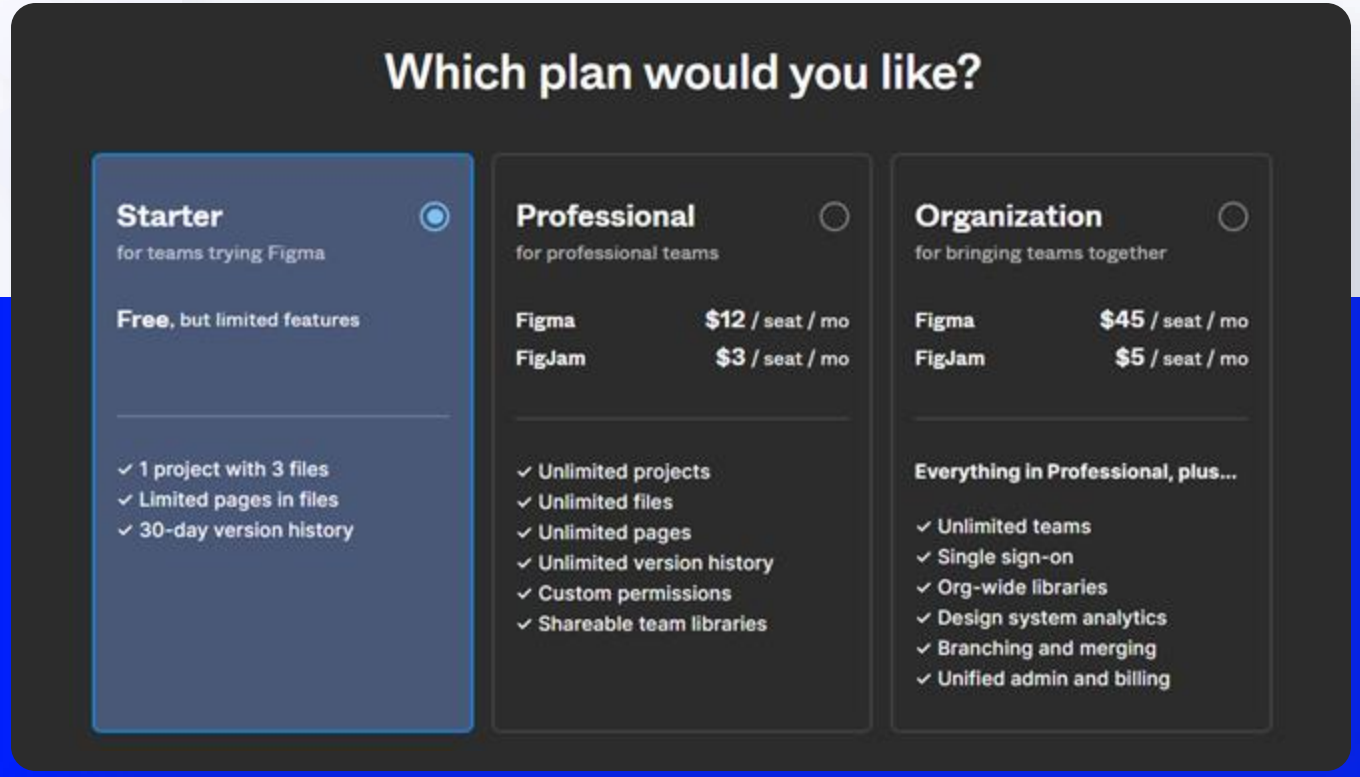
What is Figma / Why it has Become Popular

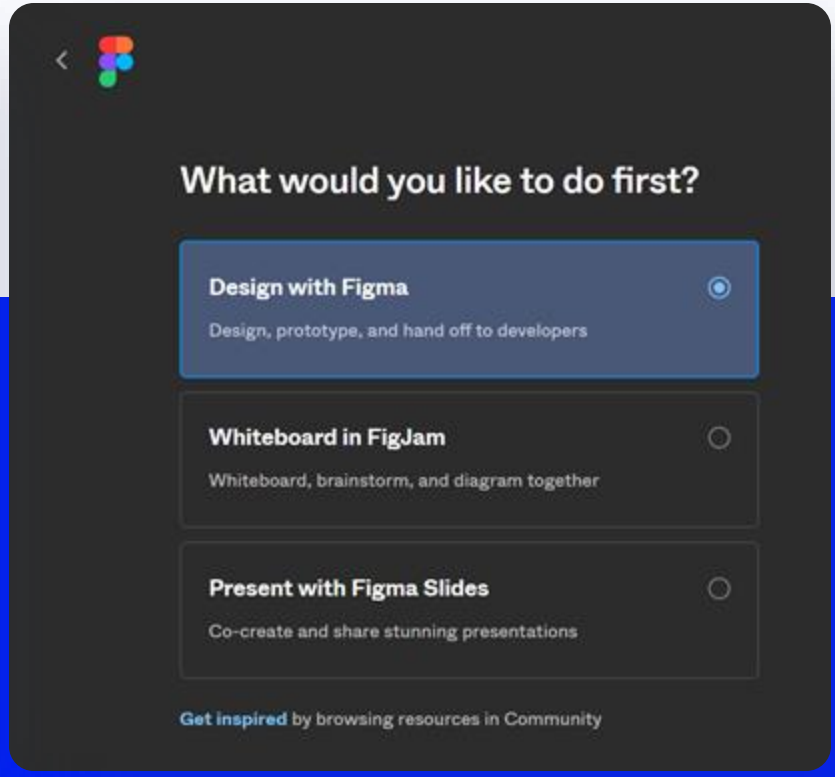
- Figma is a design / prototyping tool that can be easily shared with multiple team members to enable collaboratively working together on the same design.
- Notes/feedback from collaborators helps the designer /design teams to get and incorporate feedback easily.
- Figma can provide CSS code for UI elements (eg. fonts, colours, sizes)
- Figma is a cloud-based tool which allows designers to work remotely, in both a desktop app & web app. It also has an autosave function.
- Figma can be used for free for an individual.
- Figma provides free membership for students or educators.

Most enterprises make use of Figma. It is industry standard and highly recognized in the designers community as well.

Figma Editing Tools

Sign up and create a free individual Figma account at https://www.figma.com/. **Verify** your email address and complete the **profile questions**, then select the **Starter** free plan, and choose to start out by Designing with Figma.



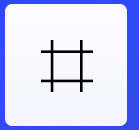


Figma Design Basics

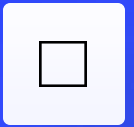
Figma is fairly intuitive, with many similarities to other design tools, but there are some key terms to be familiar with (see here for more):

Canvas: the design backdrop, which holds frames, shapes, text, and images. It spans about 65,000 points in all directions, with a default colour of #E5E5E5.

Frame: Figma's version of an artboard, serving as containers for shapes, text, and images, representing **individual screens** in your design. Use the frame tool from the toolbar and frame presets on the sidebar to select frame dimensions for mobile or desktop sized screens.



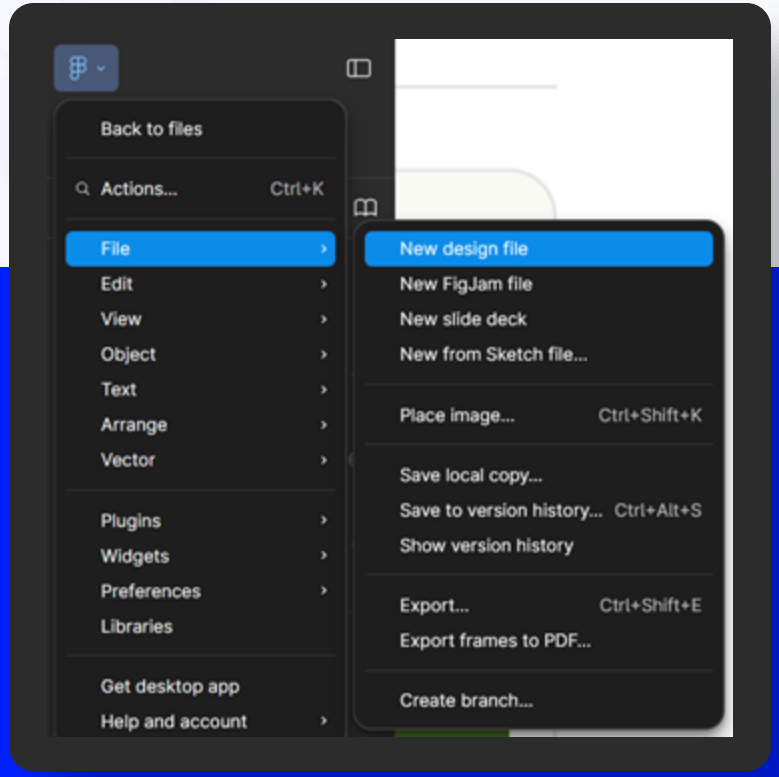
Shapes: Rectangles, lines/arrows and ellipses are basic **design elements** used to represent components in a **frame**. They can be positioned and resized manually by dragging or adjusted precisely using X, Y, width, and height fields in the sidebar.

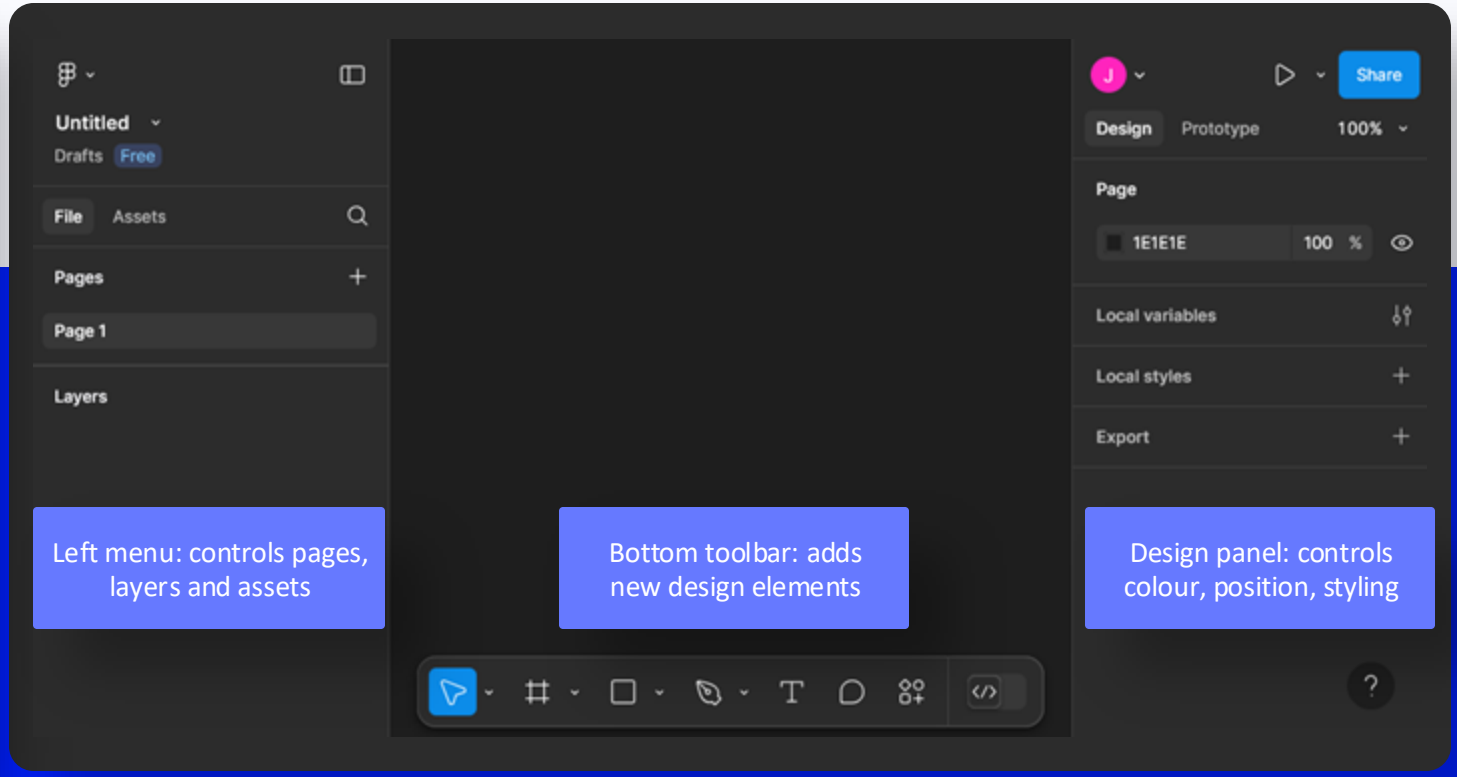


Figma Lo-Fi Prototype

Get started by creating a new design file. Use the left menu and go to **File > New design file**.

This will open a new *Untitled* blank canvas, with the toolbar at the bottom to let you start adding elements to it.





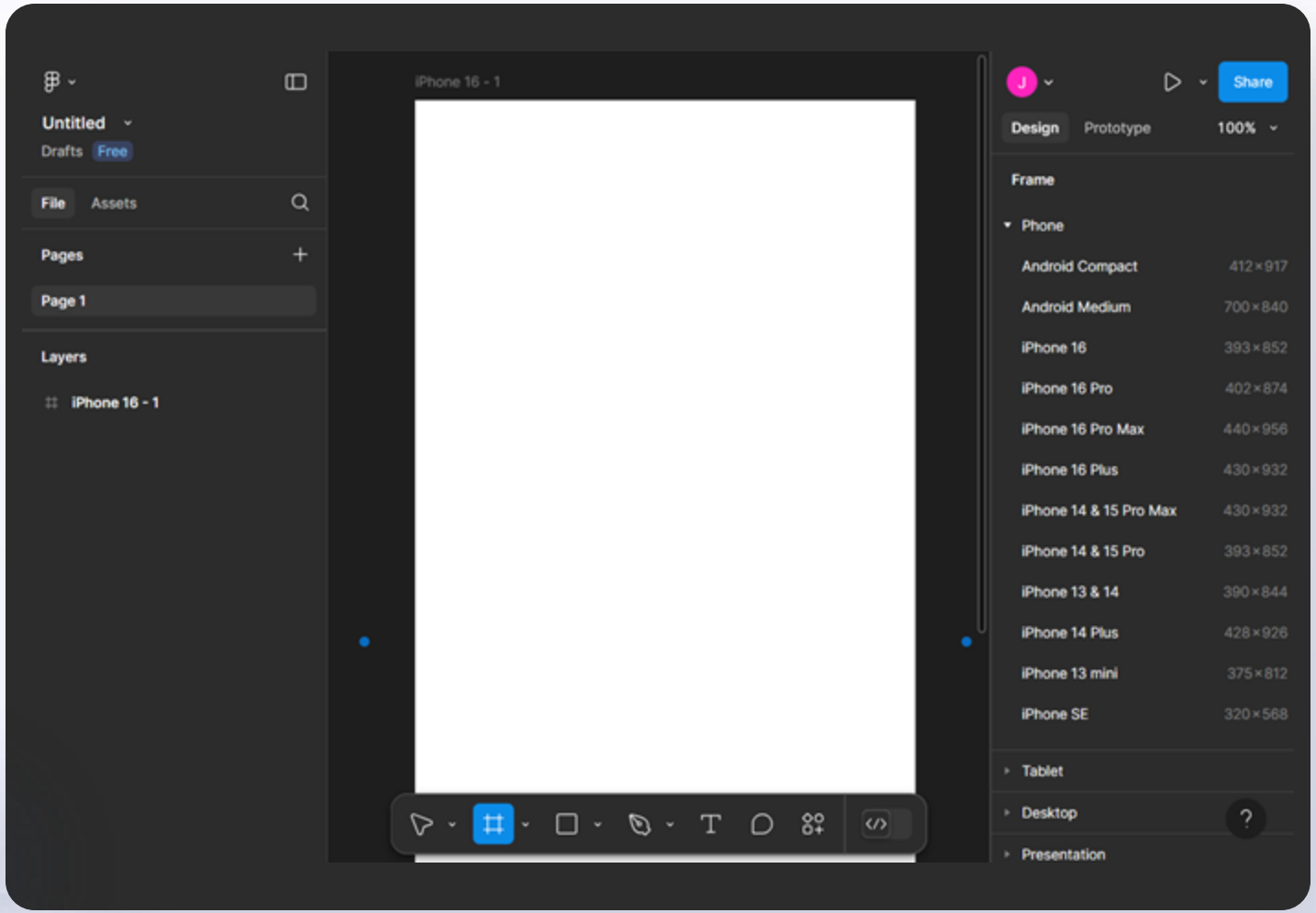
Figma Lo-Fi Prototype

Add a new **Frame** using the bottom toolbar and select a size preset (eg. iPhone 16).

Then use the **Shape** tool to add rectangles, circles and lines to your design.

The **Text** tool will allow you to include text items as well.

***Tip**: hold the spacebar while you drag with the mouse to move around your canvas.*

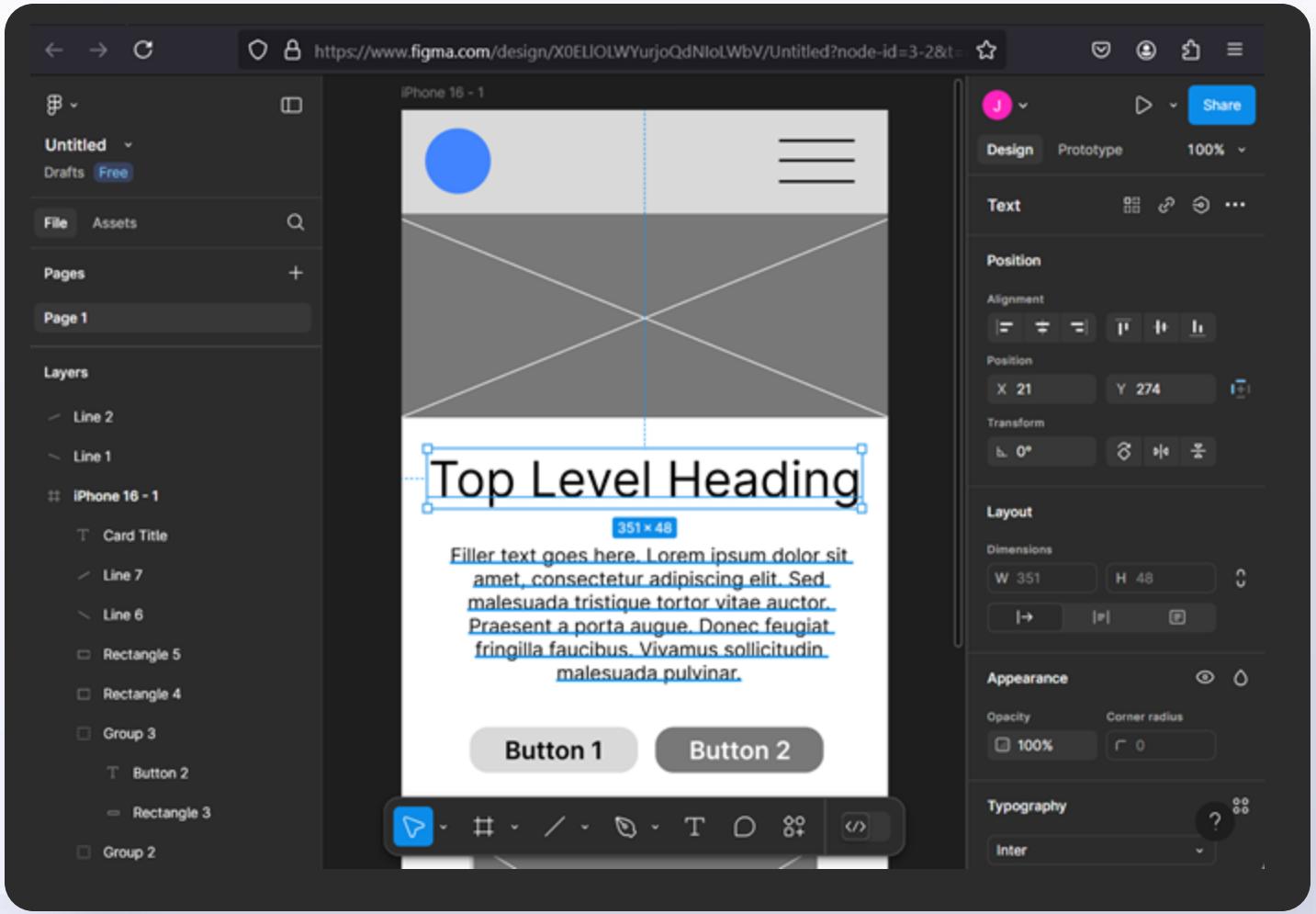


Figma Lo-Fi Prototype

After using the bottom toolbar to add some elements to our design, we can use the right **Design** panel to tweak things like the alignment, position, orientation, opacity, typography, and colours.

The left **Layers** panel lists all the elements individually.

Right-clicking any element or layer will bring up an additional context menu.



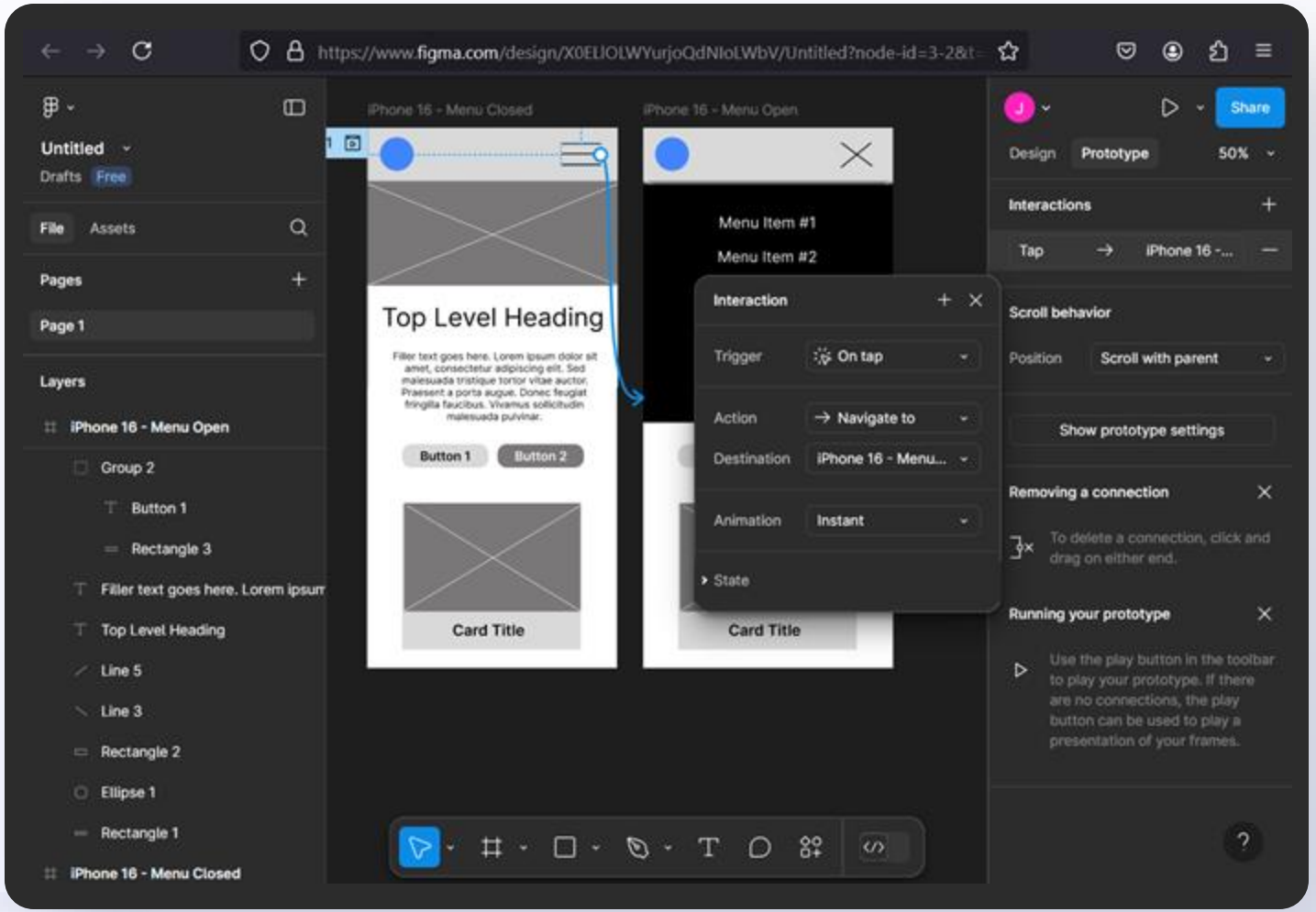
Figma Lo-Fi Prototype

With multiple frames, we can add **interactivity** to our design by simulating how a user progresses from one screen (frame) to another, turning it into a **prototype**.

Switch from the **Design** panel on the right to the **Prototype** panel.

Click on an element with interactivity (such as the menu hamburger), and add an **Interaction**.

In this case we want to **navigate** to the **Menu Open** frame when the **hamburger** element is **tapped**.

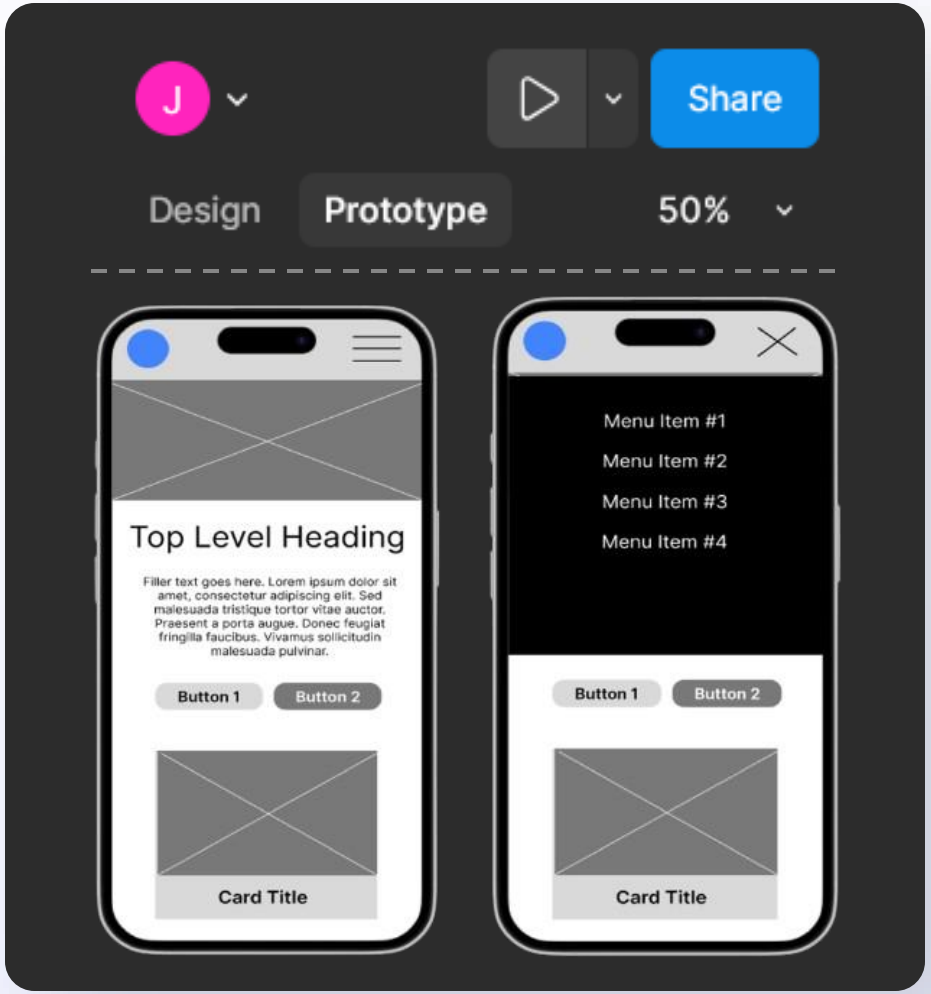


Figma Lo-Fi Prototype

To run your prototype, click on the **Present** (Play) icon above the Portfolio panel. You can now click on the interactive areas and transition between frames, simulating the flow of a real application.

Share your prototype by clicking the Share button, setting link access to "Anyone with the link (can view)," and copying the link to share with your team.

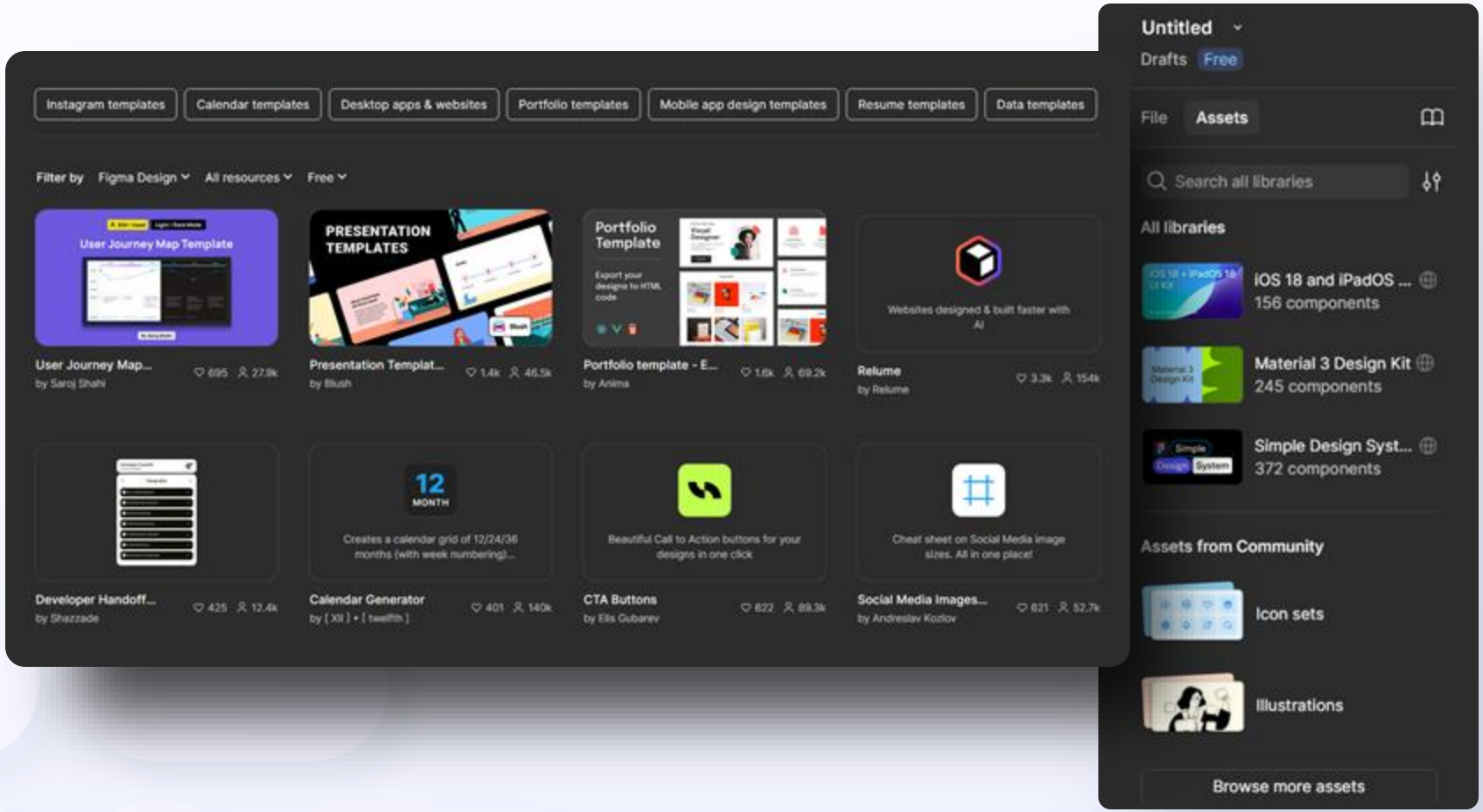
Use the **Comment** tool to provide feedback directly on the prototype. Comments are visible on frames in the editor, allowing teams to collect, review, and resolve feedback efficiently.



Figma Lo-Fi Prototype

Figma also includes both paid and free community resources to use in your projects.

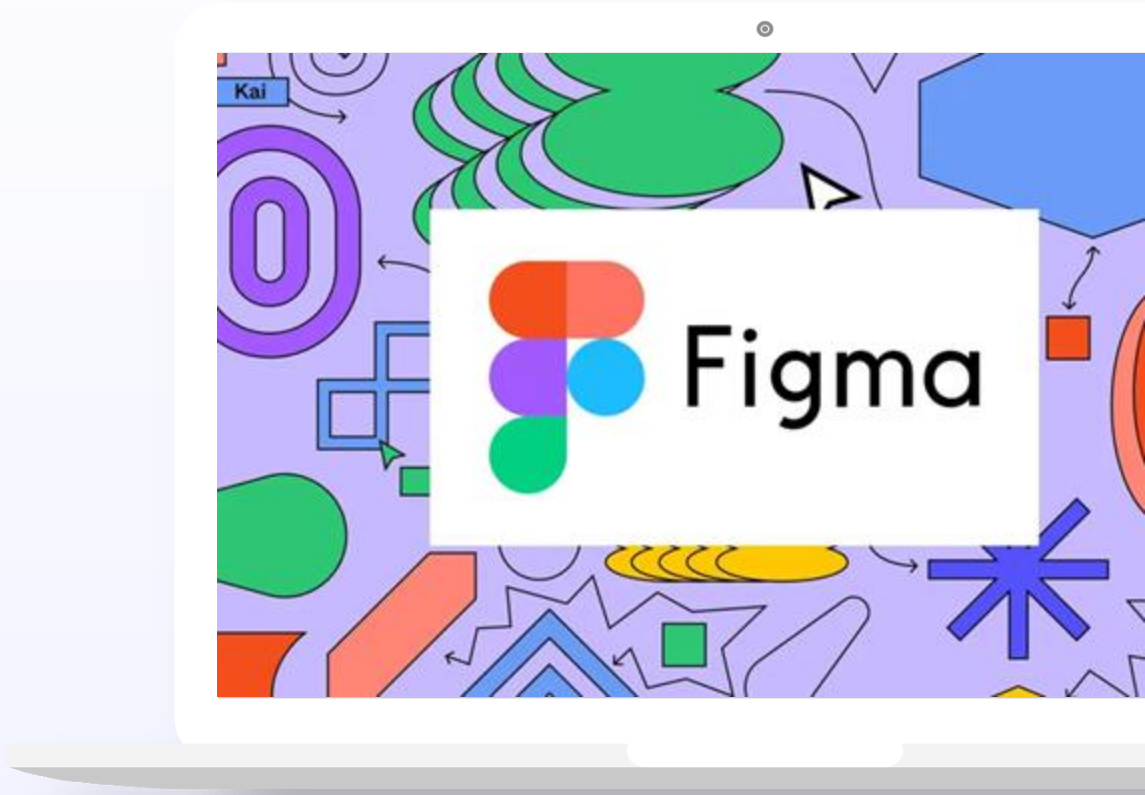
Click the **Assets** tab on the left panel to view the default **libraries**, icon sets and **illustrations**, or click on **Browse more assets** to view a larger library of templates and resources for all kinds of projects:



Lab #1: Figma

Create a Figma prototype for a social media application, where you can post your content and posts from other users are visible. Don’t just jump into Figma, try to follow the design procedure by first clarifying the requirements, understanding the application flow, and sketching a quick paper mockup.

Also, this could be part of your portfolio, so try to do some research and come up with a personal design. For example, it could be a social media application dedicated to the Warhammer Community, or the Ferrari Lovers.



What are frameworks in Javascript?

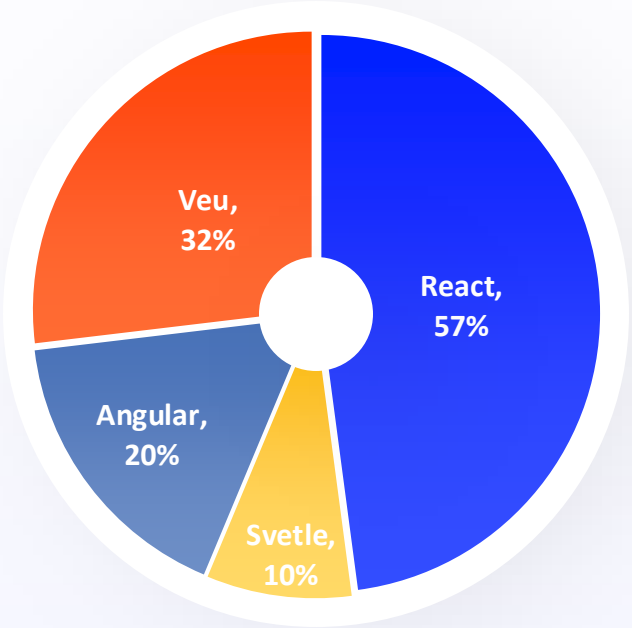
Frameworks act as "**scaffolding**" for building larger applications. **JavaScript**, often criticized for its hasty two-week development, has inspired the creation of numerous frameworks to "**abstract**" modern functionalities, making development more robust and efficient. For instance, JavaScript lacks a efficient way to repeatedly update the UI dynamically, a problem frameworks like **React** address effectively. Applications that do not use any additional library or framework are commonly referred to as "**Vanilla JavaScript**."

Today the most popular frameworks are **React**, **Angular** and **Vue**, but many others exist, with pros and cons. React is a UI library, Angular is a fully-fledged front-end framework, while Vue.js is a progressive framework. In this program we will focus on **React**.

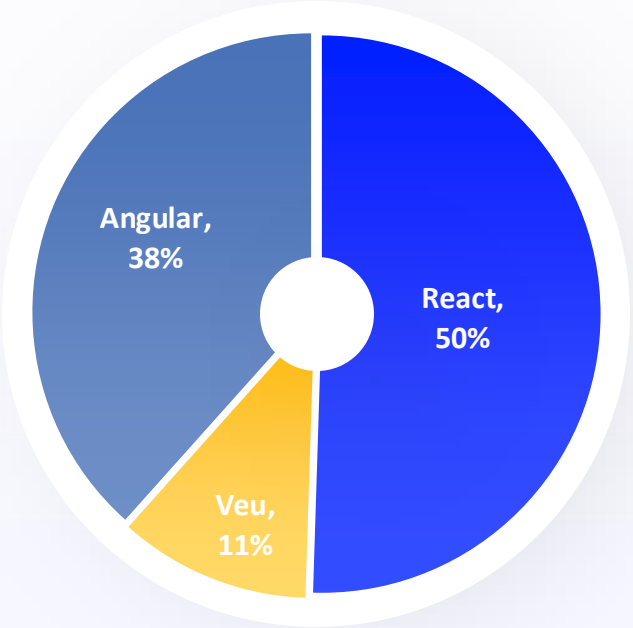
The Job Market

When entering the job market, it is important to look at the trends, but remember, this may change very quickly. What is trendy today may not exist tomorrow.

As a person in IT, take these numbers with a pinch of salt. Also, it is very normal for a good software engineer to be requested to jump from one framework to another, or even change programming languages altogether.



Percentage of developers using major JS frameworks in 2024, as reported by voluntary developer surveys



Percentage of major JS frameworks required by frontend developer job descriptions in 2024

React

React is the least complex of the three frameworks. That is because you only need to import the library, then you can start writing your React application with a few lines of code. There are also several bundler tools which make starting a new React project easier by handling common configuration.

Most React applications are component-based and render multiple components and elements on the page.

You can start using React with just a few lines of code.

```
ReactDOM.render(
<h1>Hello, world!</h1>,
document.getElementById('root’)
);
```

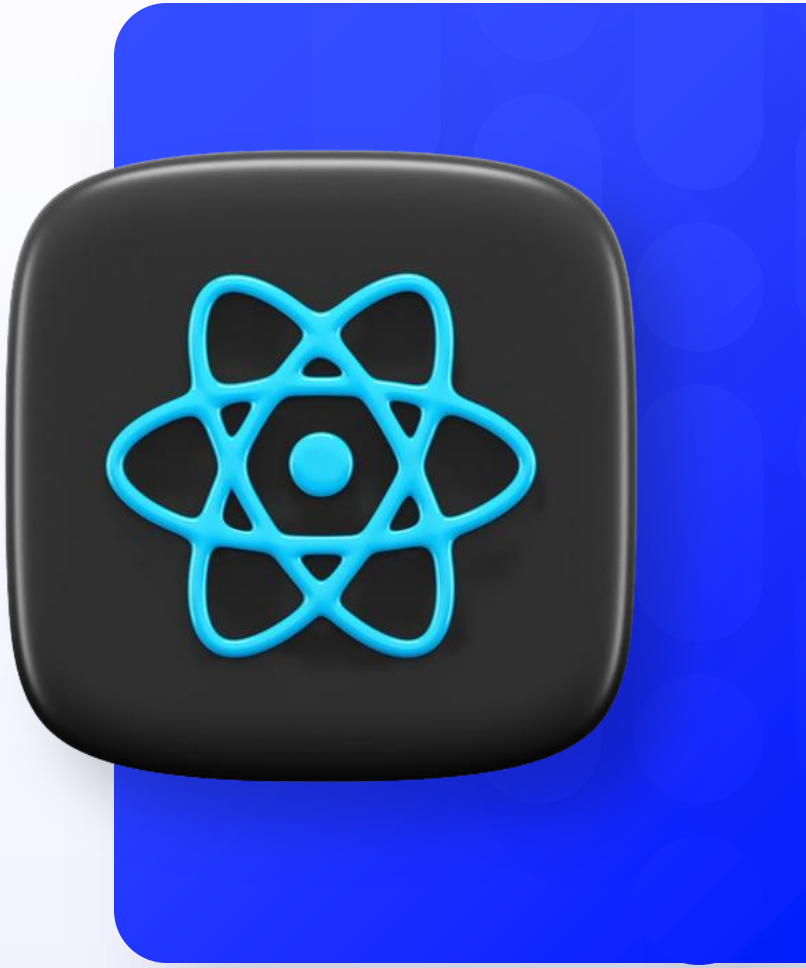
React Continued

React Elements are the smallest building blocks of React apps. They are more powerful than DOM elements because the React DOM makes sure to update them efficiently whenever something changes.

Components are larger building blocks that define independent and reusable pieces of markup to be used throughout the application. They accept inputs called props and produce elements that are then displayed to the user.

React is based on JavaScript, but it’s mostly combined with JSX (JavaScript XML), a syntax extension that allows you to create elements that contain HTML and JavaScript at the same time. Anything you create with JSX could also be created with the React JavaScript API, but most developers prefer JSX because it’s more intuitive.

React Native is an extension that is used for developing mobile applications.

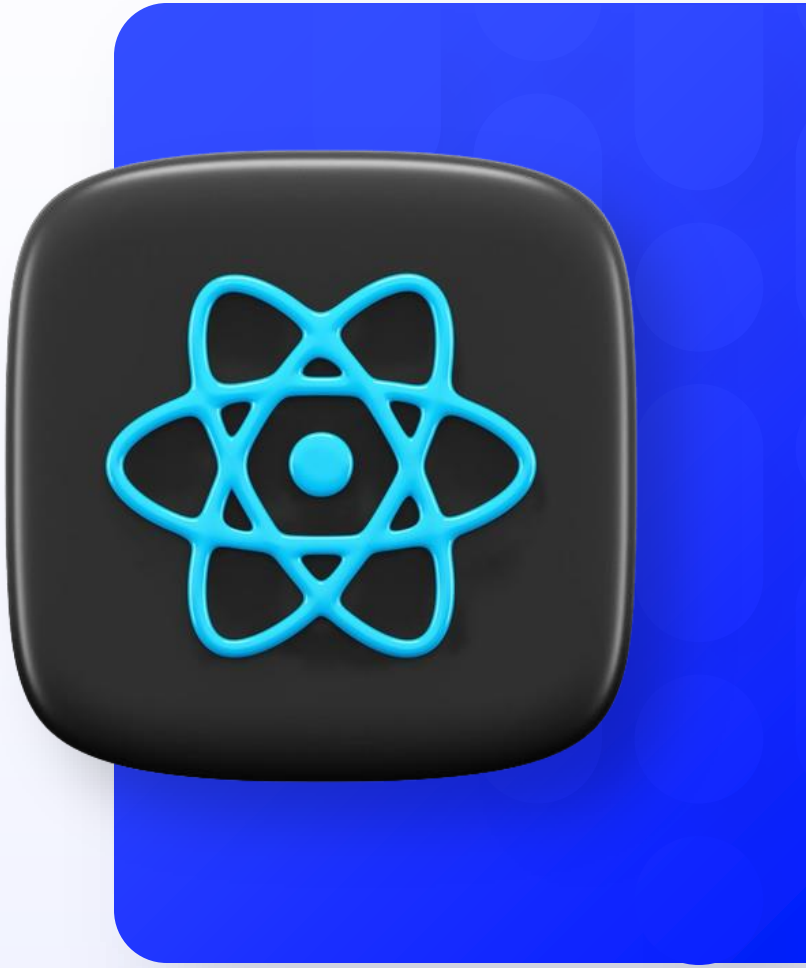


React Problems

React is a **client-side rendering** framework, which means all data are handled and rendered on the client's browser. This causes problems when:

- The amount of data becomes too large (e.G. 10000 items to be displayed as a list).
- The application/website needs to be found by search engines (because js-generated pages traditionally have low indexing and ranking).
- There are too many components in the app, because re-rendering management becomes difficult to make sure only needed components are re-rendered.

Since React is so widely used and well-supported, these problems are increasingly being addressed by updates and improvements in recent versions. Extension frameworks such as **Next.js** also use server-side rendering to mitigate these issues.



Angular

Angular has the most complex project structure out of the three, and since it's a fully-blown front-end framework, it relies on more concepts.

Since Angular works best with **TypeScript**, it's important that you know TypeScript when building an Angular project. TypeScript is similar to JS but requires that all variables are defined with and retain a specific data type.

Similar to React, Angular is also component-based, BUT it provides built-in solutions for state management, routing, form handling, dependency injection, and HTTP services out of the box, while React requires additional libraries for these features.

Angular Continued

Angular is designed with enterprise-scale applications in mind, offering built-in features for testing, performance optimization, and modularity.

Projects in Angular are structured into **Modules**, **Components**, and **Services**. Each Angular application has at least one root component and one root module.

Each component in Angular contains a **Template**, a Class that defines the application logic, and **MetaData** (Decorators). The metadata for a component tells Angular where to find the building blocks that it needs to create and present its view.

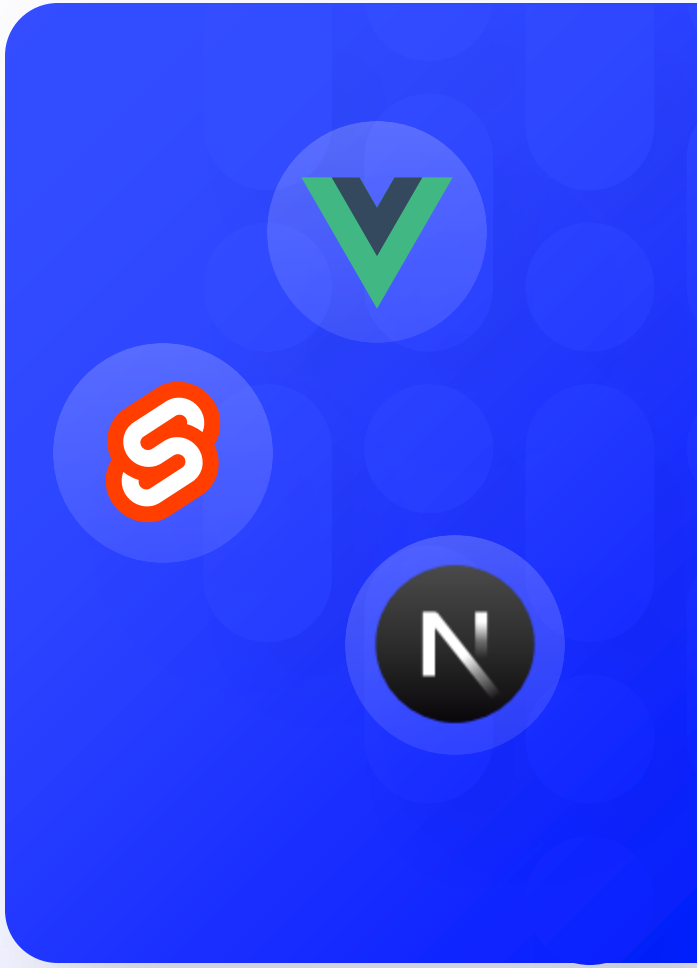
This **Model-View-Controller** (MVC) style of framework is also used heavily outside of Angular and we will look at it further in this and subsequent modules.

Other Frameworks

Vue.js : An increasingly popular component-based, **progressive** JS framework that focuses on approachable syntax, built-in reactivity, and seamless integration into existing projects, offering a balance between simplicity and feature depth without the boilerplate often found in React or Angular.

Svelte : A modern, open-source JS UI framework. Known for its innovative approach, it focuses on compiling components at **build time** rather than relying heavily on a virtual DOM during runtime, as seen in frameworks like React or Vue.

NextJS : Based on the React library. Next.js is a **server-rendered** React framework that offers simplified routing and back-end integration while still supporting standard React component-based UIs.



No Framework - Vanilla JS

Simplicity and Control: Vanilla JS gives you direct interaction with the DOM and browser APIs, avoiding the abstractions and "magic" of frameworks.

Better Performance: No dependency overhead or extra code, resulting in smaller bundle sizes and faster load times.

No Dependencies: No need for third-party libraries or build tools, reducing complexity, security risks, and versioning issues.

Long-Term Stability: Unlike frameworks that may lose support, vanilla JS relies on evergreen web standards, ensuring maintainability.

Vanilla JS is ideal for small projects, fast prototyping, and teams seeking a lightweight, dependency-free solution. A solid understanding of client-side JS without a framework is essential for best results when using a framework.

Lab #2 - Calculator

In this exercise you will need to create a calculator.

Requirements:

The application should take 2 numbers, and support 4 operations (+, / , x , -) .

You need to press the equals button to get the result displayed, and reset to clear it.

NOTE: Do not use the eval() function - it is a security risk and bypasses custom logic.

1. List the requirements, in this case you have a total of 4 requirements :
   a) Get data (two numbers)
   b) Choose an operator
   c) Get the result
   d) Reset the screen
2. Sketch the application, so that you are sure about the correct functioning.
3. You may use a flow diagram to help.
4. Use a tool of your choice, like Figma, to design the application.
5. Use the prototype ability and test it.

Develop the application. Start from GIT - it is good practice to do things in a standard way.

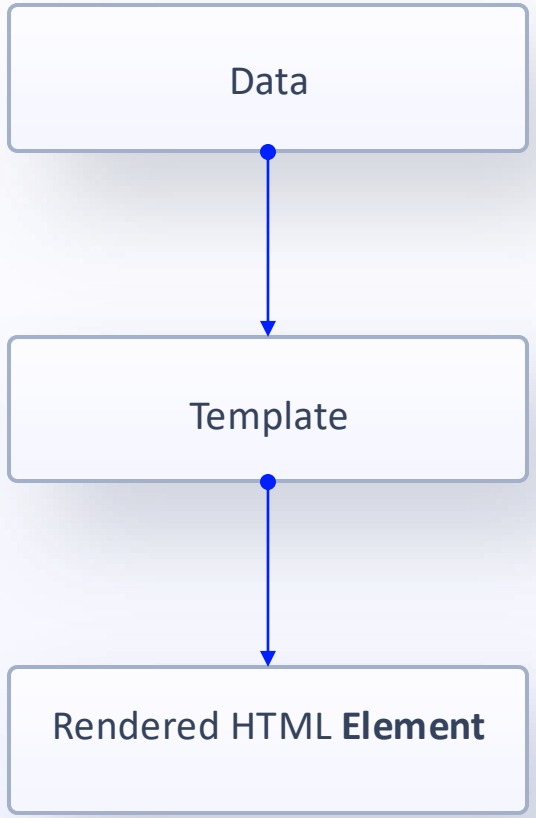
- Create a repository
- Clone the repository locally
- Create a branch for each feature

Templating

In the last 20 years, web development has become much more data-oriented. For this reason, we have moved towards the **MVC** model (**M**odel **V**iew **C**ontroller).

To simplify this, the various frameworks like React and Angular have become data-centric. To satisfy this **data-centrism**, we have moved towards the concept of **Templates**.

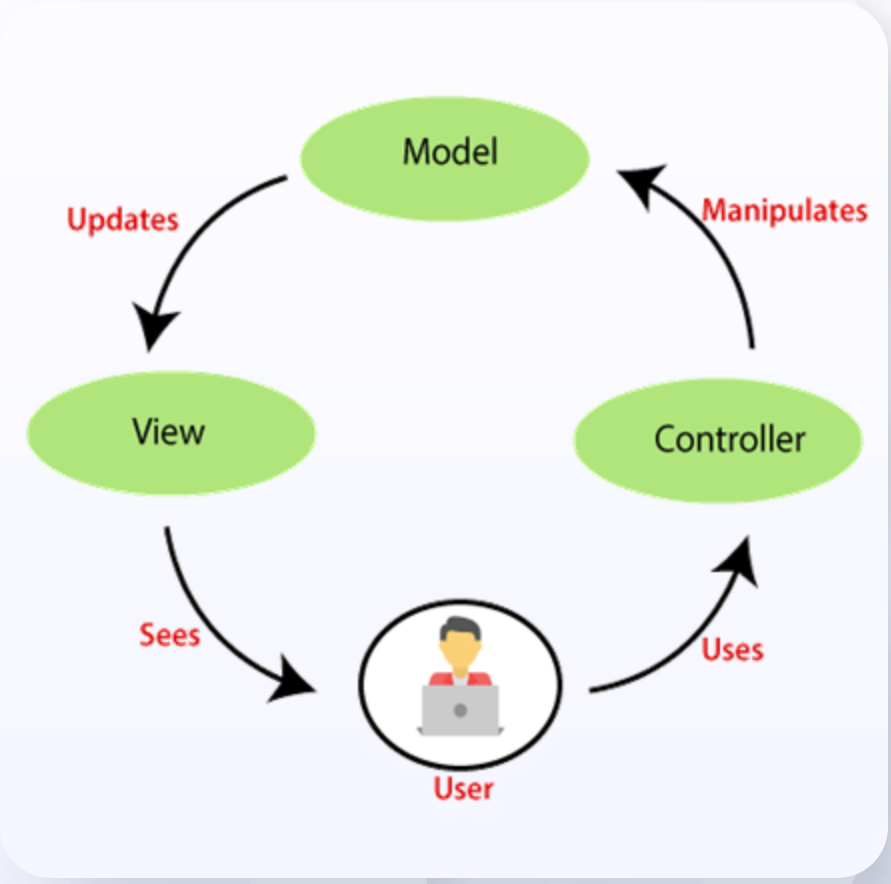
Templates are **reusable** pieces of code that are centred around the format of the **data**. They take raw data and display it in a consistent, user-friendly, formatted, reusable way.



Short intro to MVC

The MVC model is everywhere and we will revisit it in several modules. Try to get a high level understanding, it is not straightforward the first time you see it.

- Model: Our data is defined in this section. It's where we save our schemas and models, or the blueprint for our app's data.
- View: This includes templates as well as any other interaction the user has with the app. It is here that our Model's data is given to the user.
- Controller: This section is where the business logic is handled. This includes database reading and writing, as well as any other data alterations. This is the link between the Model and the View.$ git clone link to repo



HTML Template

The HTML `<template>` element represents a template in your markup. It contains "template contents"; essentially chunks of cloneable DOM. Templates act as pieces of scaffolding that you can use and reuse throughout the lifetime of an app.

To create templated content, declare some markup with placeholders for your data, and wrap it in the `<template>` element with a unique id:


```
<template id="mytemplate">
    <img src="" alt="great image">
    <div class="comment"></div>
</template>
```

JavaScript DOM API

Once we have our template - or **'view'** from the MVC framework - we can use JavaScript to define (or obtain) the data - or **'model'** - that will feed into it, and to handle the steps required to repeatedly **clone** the template, **populate** the template with our data, and then **include** the newly populated template back into the page.

This process will require some manipulation of the HTML Document Object Model (DOM) rendered by the browser, so it is handy to review some built-in DOM API functions.

In particular, functions such as document.getElementById and document.querySelector are useful for **accessing HTML elements** in order to read and modify the innerText and innerHTML properties. appendChild is useful for **adding new or updated elements** into the DOM at specific locations.

Custom Templates

Start by creating a template for a card together with the CSS required for the template:

You can do this using more than one template in the same HTML file.

This template represents a single card. We can duplicate it as many times as we need, using javascript.

Copy this code and store in cards.html.

Creating a template :

Side is a template for a card with the CSS written alongside.

```
<!DOCTYPE html>
<html>
   <head>
       <style>
           body {
             font-family: Arial, Helvetica, sans-serif;
             background-colour: #e3f2fd;
           }
           .card {
             padding: 10px;
           }
           .card-title {
             font-weight: 600;
             font-size: 3em;
             padding: 0 0 10px 0;
           }
           .card-description {
             font-weight: 400;
             font-size: 2em
           }
       </style>
   </head>
   <body>
       <template id="card-template">
           <div class="card">
             <div class="card-body">
               <div class="card-title"></div>
               <div class="card-text"></div>
             </div>
           </div>
       </template>

       <div id="card-list"></div>
   </body>
</html>
```

Custom Templates

In the same file, you can add your JavaScript.

Then JavaScript can be used to add one or many cards, all with the same template but different data.

Copy this <script> section and include it just before the closing </body> tag in cards.html.

Creating a template :

Side is a template for a card with the CSS written alongside.

```
<script>

    function addCard() {
       // clone the template
       const template =
          document.getElementById("card-template")
                  .content.cloneNode(true);

       // populate the template
       template.querySelector('.card-title').innerText =
           'My Card Title';
       template.querySelector('.card-text').innerText =
           'lorem ipsum ble bla';

       // include the populated template into the page
       document.querySelector('#card-list')
               .appendChild(template);
    }

    addCard();
</script>
```

Lab #3 - Templates

Exercise 1 :

Modify the **addCard** function from the previous slide so that you can pass content for the card dynamically.

Exercise 2 :

Call **addCard** repeatedly so that your cards are automatically generated based on data from an array. This way you will create as many cards as you need to display all the data in the array.

Exercise 3 - the artist's portfolio:

Populate the page dynamically, by generating an artist profile card which includes cards representing the items in an artist's portfolio. *Extension: make an array of artists, all with multiple portfolio items.*

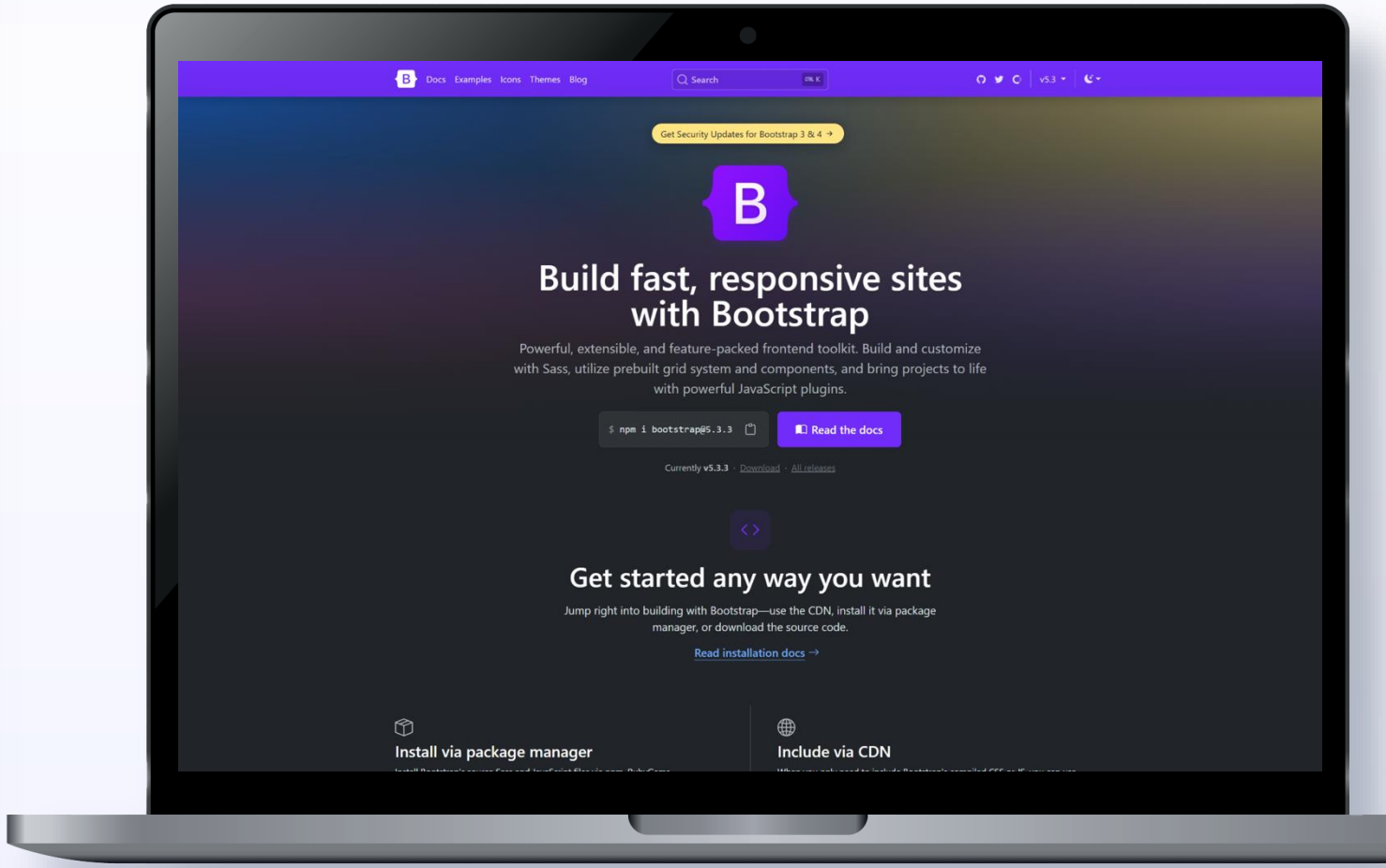
For Exercise 2 use the following array :

```
const data = [{name: 'bob', age: 23}, {name: 'alice', age: 39}]
```

For Exercise 3 use the following:

```
const artist = {
    name: "Van Gogh",
    portfolio:[
        {title: "portrait", url:
"https://collectionapi.metmuseum.org/api/collection/v1/iiif/436532/1671316/main-image"},
        {title: "sky", url: "https://mymodernmet.com/wp/wp-content/uploads/2020/11/White-house-night-van-goh-worldwide-2.jpg"},
    ]
}
```

BOOTSTRAP


Docs Examples Icons Themes Blog
Search
Get Security Updates for Bootstrap 3 & 4 →
Build fast, responsive sites with Bootstrap
Powerful, extensible, and feature-packed frontend toolkit. Build and customize with Sass, utilize prebuilt grid system and components, and bring projects to life with powerful JavaScript plugins.
$ npm i bootstrap@5.3.3
Read the docs
Currently v5.3.3 · Download · All releases
Get started any way you want
Jump right into building with Bootstrap—use the CDN, install it via package manager, or download the source code.
Read installation docs →
Install via package manager
Include via CDN

Enter Bootstrap

Have you ever wondered why so many websites all look the same?

Bootstrap is a CSS framework based on HTML, CSS and Javascript. It is used for developing responsive layout and mobile first web projects. It includes:

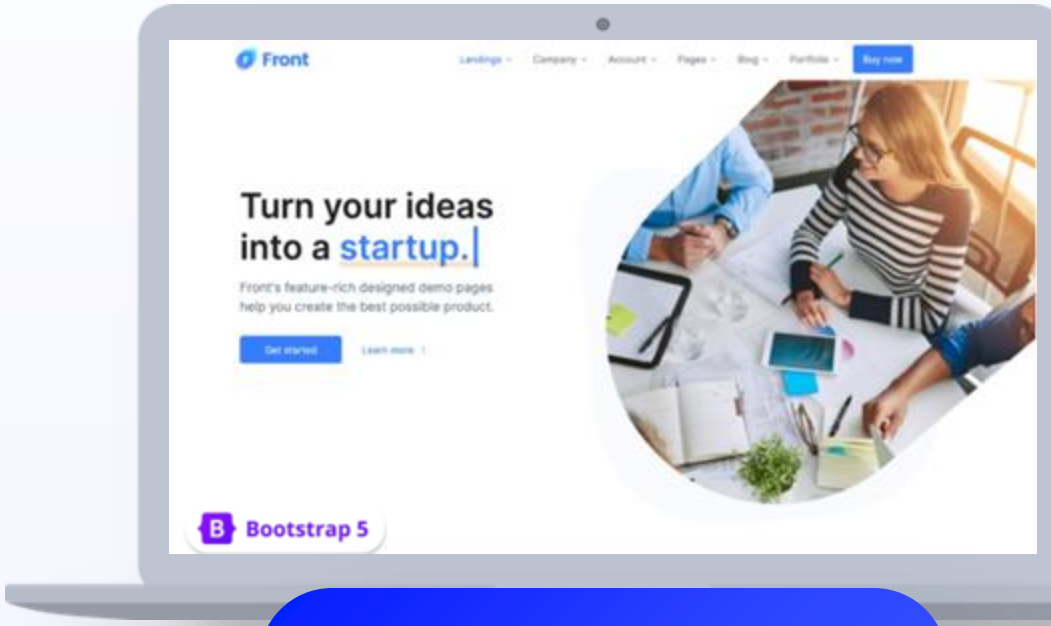
Basic structure with containers, a grid system, and general utility classes.

CSS: Global CSS settings, the definition of basic HTML element styles, extensible classes, and an advanced grid system.

Components: Bootstrap includes a lot of reusable components for creating navigation, menus, cards, pop-up boxes and more.

JavaScript plugins: Bootstrap includes custom jQuery plugins. We can use all the plugins directly, or we can use them one by one.

Customisation: we can customize Bootstrap components to get our own version.



See Bootstrap for more info and tips

Bootstrap Basics

Here are the steps:

- For simplicity, use the CDN to include both CSS and JS. Get the most up-to-date version from Bootstrap
- Add the **CSS** CDN link to the head tag of the html file.
- Add the **JS** CDN link to the bottom of the body tag of the html file.
- Add the **viewport** meta tag to the head of the page.
- Grab some starter code from Get started with Bootstrap or to the right:

```
<!doctype html>
<html lang="en">
<!-- use the most recent version →
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1">
    <title>Bootstrap demo</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/css/
bootstrap.min.css" rel="stylesheet" integrity="sha384-
4bw+/aepP/YC94hEpVNVgiZdgIC5+VKNBQNGCHeKRQN+PtmoHDEXuppvnDJz
QIu9" crossorigin="anonymous">
</head>

<body>
    <h1>Hello, world!</h1>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/js/bo
otstrap.bundle.min.js"    integrity="sha384-
HwwvtgBNo3bZJJLYd8oVXjrBZt8cqVSpeBNS5n7C8IVInixGAoxmnlMuBnhb
grkm"
        crossorigin="anonymous"></script>
</body>
</html>
```

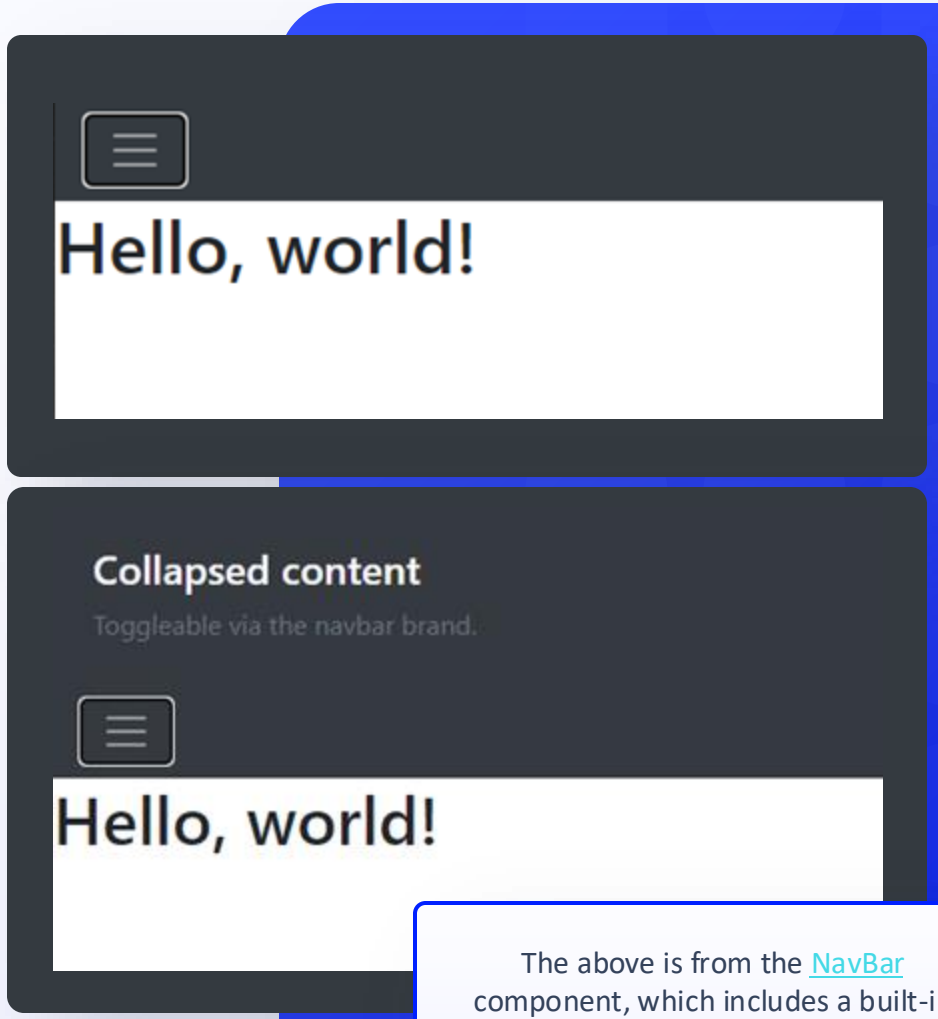
Simple But Effective

With very little effort you can create great looking pages using the standard components. Using a mix and match to combine bootstrap with your own custom styling is even better!

Bootstrap is perfect for beginners, but can become as complex as you want it to be.

Explore more components and play around with them.

See : Components



The above is from the NavBar component, which includes a built-in mobile menu toggle.

Bootstrap CSS

Once we import the CSS library to a page, it will use the default CSS styles from bootstrap. Remember, these are default, but they can still be customised using normal CSS rules.

```
<button type="button" class="btn btn-primary">Bootstrap Button</button>
```

It will display a button with blue background and white text. On the right is a standard html button, you can see the improvement. Other styles can be used as well, just by choosing the desired classes. See: Buttons

Bootstrap Layouts

The grid system is built with a flexbox layout that uses containers, rows, and columns to lay out and align content, and is fully responsive. See: Grid System The grid system divides a row into **12 columns**.

So we can display rows like this, by dividing the total 12 columns into various combinations:

```
<div class="row">
    <div class="col-12">This is a whole row</div>
</div>
```

```
<div class="row">
    <div class="col-2">2 columns</div>
    <div class="col-10">10 columns</div>
</div>
```

Built-in bootstrap classes can also define different column widths for different sized devices.

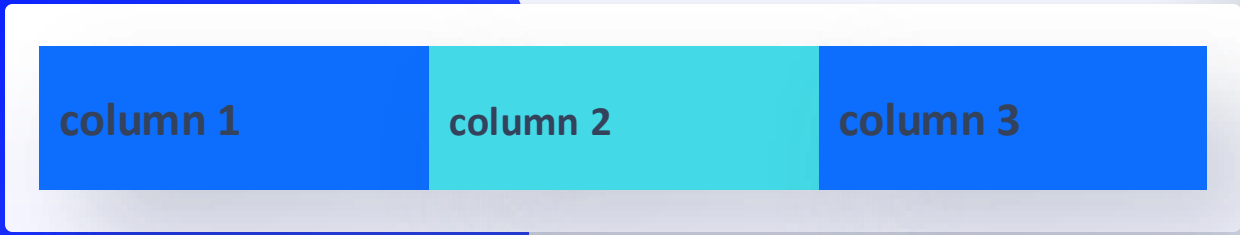
Bootstrap Layouts

Equal width columns

If we have not set the width in columns for an element with a “col” class, all columns in the row will be the same width.

An example of dividing a row into 3 same-width columns:

```
<div class="row">
    <div class="col bg-primary">column </div>
    <div class="col bg-info">column 2</div>
    <div class="col bg-primary">column </div>
 </div>
```



Responsive Design in Bootstrap

Bootstrap has default breakpoints. It uses modifiers (-xs, -lg etc) representing different widths for different screen sizes (we can customize it as well).

X-small screen(*-xs) : less than 576px (phones)

Small screen(*-sm) : 576px and up

Medium(*-md) : 768px and up (tablets)

Large(*-lg) : 992px and up (small laptops)

X-large(*-xl) : 1200px and up (laptops)

Xx-large(*-xxl) : 1400px and up (large screens)

See: Breakpoints

Responsive Container:

```
<div class="container-fluid bg-danger"> alway 100% wide </div>
<div class="container-sm bg-primary">100% wide until small breakpoint</div>
<div class="container-md bg-info">100% wide until medium breakpoint</div>
<div class="container-lg bg-success">100% wide until large breakpoint</div>
<div class="container-xl bg-danger">100% wide until extra large breakpoint</div>
<div class="container-xxl bg-warning">100% wide until extra extra large breakpoint</div>
```

Responsive Design in Bootstrap

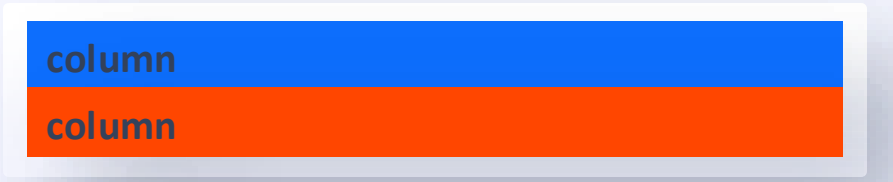
Responsive Columns

For responsive columns, we can add “col-md-*/col-sm-*” etc, to indicate different column widths at different breakpoints.

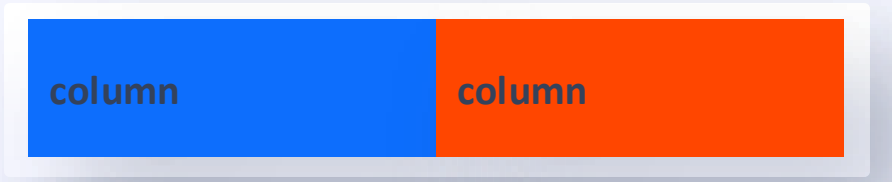
For example, if we want to display a column as taking up a whole row (all 12 columns) when the screen is small, otherwise displaying as two equal-width (6 out of 12) columns:

```
<div class="row">
      <div class="col-12 col-sm-6 bg-info">column</div>
      <div class="col-12 col-sm-6 bg-danger">column</div>
  </div>
```

- **On small screens (576px) and below:**



- **On widths more than a small screen:**



Bootstrap components

Bootstrap has **HTML/CSS** components and **JS** components.

HTML/CSS components like **button** / basic **card** can be used by **finding** a close example of your desired layout from the Bootstrap website, **copying** the example code, and then **customizing** the content and default styles. Even JS-based components such as **Modal**, **Alert**, and **Carousel** are simple to copy and adapt.

The major components of bootstrap:

Buttons, Card, Carousel, Dropdowns, Forms, Input Group, List Group, NavBar, Modal, Navs and tabs, Pagination.

Try including several of the above into your own page!

Example of Modal:

We can simply add a trigger to the button to display the corresponding modal box.

```
<button type="button" class="btn btn-primary" data-bs-
toggle="modal" data-bs-target="#exampleModal">
   Launch demo modal
</button>

<div class="modal fade" id="exampleModal" tabindex="-1"
aria-labelledby="exampleModalLabel" aria-hidden="true">
   <div class="modal-dialog">
       <div class="modal-content">
          <div class="modal-body">
            hello modal
          </div>
        </div>
   </div>
</div>
```

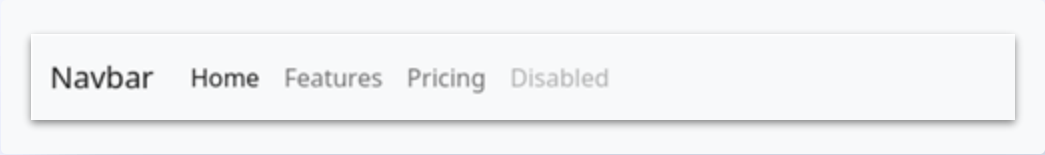
Bootstrap components

The navigation bar is one of the most important interface elements for your application, as it directs and drives the user.

Making the right decisions about what's included will allow users to find things quickly. Try to balance functionality with appearance. You don't want to create complex nested menus, but also, you don't want your user to spend too much time looking for things.

Use clear, concise labels and prioritize essential links to avoid overwhelming users. Bootstrap ensures other key UX concerns such as accessibility and responsiveness.

See Navbar



Example :

Navigation Bar - NavBar

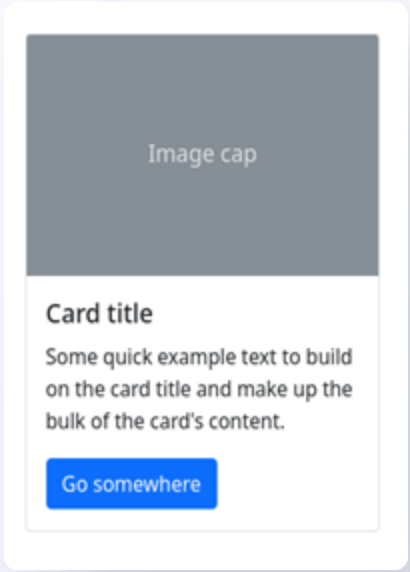
```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <div class="container-fluid"> <a class="navbar-brand" href="#">Navbar</a>
<button class="navbar-toggler" type="button"
      data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-
controls="navbarNav" aria-expanded="false"
      aria-label="Toggle navigation"><span class="navbar-toggler-
icon"></span></button>
    <div class="collapse navbar-collapse" id="navbarNav">
      <ul class="navbar-nav">
        <li class="nav-item"><a class="nav-link active" aria-current="page"
href="#">Home</a>
        </li>
        <li class="nav-item"> <a class="nav-link" href="#">Features</a> </li>
        <li class="nav-item"><a class="nav-link" href="#">Pricing</a> </li>
        <li class="nav-item"> <a class="nav-link disabled">Disabled</a> </li>
      </ul>
    </div>
  </div>
```

Bootstrap components

Cards are great versatile components that visually and logically group related pieces of data about a common entity.

Normally, the cards display data from objects. Take for example an online shop: product data including images, titles, prices, descriptions & links, is stored in an array, and is then iteratively rendered in cards by using the Bootstrap markup as a template, inside a Grid for responsive layout.

See Card



Example :

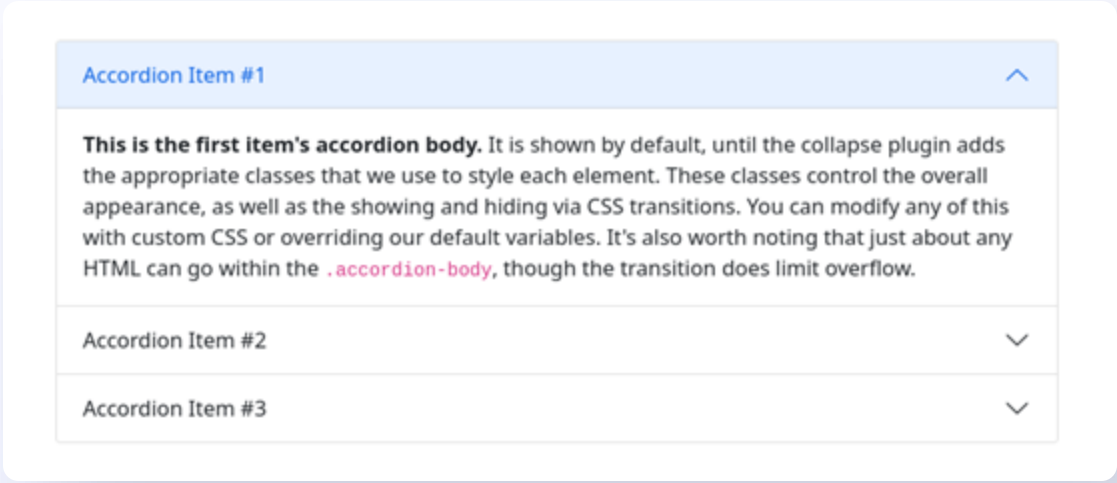
Card

```
<div class="card" style="width: 18rem;">
    <img src="..." class="card-img-top" alt="...">
    <div class="card-body">
      <h5 class="card-title">Card title</h5>
      <p class="card-text">Some quick example text to build
on the card title and make up the bulk of the card's
        content.</p>
      <a href="#" class="btn btn-primary">Go somewhere</a>
    </div>
  </div>
```

Bootstrap components

As data becomes more complex, you will need to find better ways to present it to the users.

The accordion is a good example for interactive, data-intensive explorations. It provides a data view structure that allows for closing and opening, so users can filter and view pertinent information from a large list.




```
<div class="accordion" id="accordionExample">
        <div class="accordion-item">
  <h2 class="accordion-header" id="headingOne"> <button class="accordion-button"
type="button"
          data-bs-toggle="collapse" data-bs-target="#collapseOne" aria-
expanded="true" aria-controls="collapseOne">
          Accordion Item #1 </button> </h2>
      <div id="collapseOne" class="accordion-collapse collapse show" aria-
labelledby="headingOne"
        data-bs-parent="#accordionExample">
        <div class="accordion-body"> <strong>This is the first item's accordion
body.</strong> It is shown by default,
          until the collapse plugin adds the appropriate classes that we use to
style each element. These classes
          control the overall appearance, as well as the showing and hiding via
CSS transitions. <code>.accordion-body</code>, though the transition does limit
overflow. </div>
      </div>
    </div>
    <div class="accordion-item">
      <h2 class="accordion-header" id="headingTwo"> <button class="accordion-
button collapsed" type="button"
          data-bs-toggle="collapse" data-bs-target="#collapseTwo" aria-
expanded="false" aria-controls="collapseTwo">
          Accordion Item #2</button></h2>
      <div id="collapseTwo" class="accordion-collapse collapse" aria-
labelledby="headingTwo"
        data-bs-parent="#accordionExample">
        <div class="accordion-body"> <strong>This is the second item's accordion
body.</strong> </div>
      </div>
    </div>
    <div class="accordion-item">
      <h2 class="accordion-header" id="headingThree"> <button class="accordion-
button collapsed" type="button"
          data-bs-toggle="collapse" data-bs-target="#collapseThree" aria-
expanded="false" aria-controls="collapseThree">
          Accordion Item #3</button></h2>
      <div id="collapseThree" class="accordion-collapse collapse" aria-
labelledby="headingThree"
        data-bs-parent="#accordionExample">
        <div class="accordion-body"><strong></div>
      </div>
    </div>
  </div>
```

Lab #4 – Display Cards Bootstrap

Using only Bootstrap elements, create this grid-based card layout, making it responsive.

4 cards per row on large screens (and above), 2 cards per row on medium screens, and 1 card per row on small (and below) screens.

Bootstrap Themes

Themes can help you bootstrap applications quickly by using a pre-built design.

Themes are built as an extension to bootstrap. There are many free Bootstrap themes that are ready to customise and publish, built with Bootstrap 5, MIT licensed, and updated regularly! See here for some examples.

SB Admin 2
A free Bootstrap admin theme
Free
2,539,730

Personal
A personal website theme
Free
2,407

Freelancer
A one page freelancer theme
Free
1,232,558

DATA

Dynamic Data

We live in a data-centric world, and everything can be represented as **data**. On the web, we retrieve and interpret data and display it on screen. The same data can be displayed in hundreds of ways, as everyone can make a different page to display it.

We often access data through an **asynchronous** function. In this example, we have an array of cars and we return it to the user. We use a **timeout** to give the illusion of being a real online data-fetching function.

```
const carData = [
  { title: 'Audi', description: 'Audi AG is a German automotive manufacturer of luxury vehicles headquartered in Ingolstadt, Bavaria, Germany.' },
  { title: 'Mercedes-Benz', description: 'Mercedes-Benz, commonly referred to as Mercedes, is a German luxury automotive brand.' },
  { title: 'BMW', description: 'Bayerische Motoren Werke AG, commonly referred to as BMW, is a German multinational corporate manufacturer of luxury vehicles and motorcycles headquartered in Munich, Bavaria, Germany.' }
];

function getCars() {
  return new Promise(resolve => {
     setTimeout(function(){
        // resolve the promise with the car data after 1s
        resolve(carData)
     }, 1000)
  })
}

// get data asynchronously, then console.log for testing
getCars().then((cars) => console.log(cars))
```

Dynamic Data

Using the same templating technique we have explored before, we can **iterate** over this **dynamic data**, so that we don't have to write it one by one. The size of dynamic data is often unknown in advance, so iteratively populating templates is an important MVC technique.

Given an array of cars - display them all on the screen using html `<template>` and bootstrap.

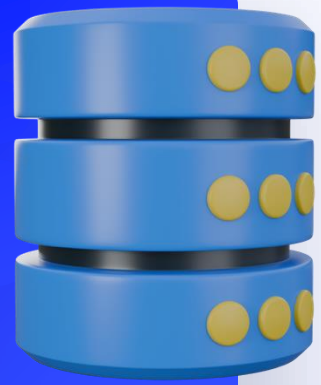
Example Code : [sanchitd5/bootstrap_example (github.com)]

```
<!DOCTYPE html>
<html lang="en">
<template id="car-template">
    <div class="card col-8" style="width: 18rem; margin:10px">
        <div class="card-body">
            <h5 class="card-title">Car title</h5>
            <p class="card-text">Car text.</p>
        </div>
    </div>
</template>
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/css/bootstrap.min
.css" rel="stylesheet">
   <title>Dynamic Cars</title>
</head>
<body>
    <div id="car-list" class="row"></div>
    <script>
     // use carData and getCars function from previous slide

     // clone, then populate, then append a new template
     function addCard(car) {
        const template = document.getElementById("car-
template").content.cloneNode(true);
        template.querySelector('.card-title').innerText = car.title;
        template.querySelector('.card-text').innerText = car.description;
        document.querySelector('#car-list').appendChild(template);
     }
     // get data asynchronously, then use it to populate a template
     getCars().then( (cars) => cars.forEach(car => addCard(car)) )
   </script>
</body>
</html>`
```

Lab #5: Manage Data 1

In this lab you will show your understanding of populating simple templates with data.

We are syncing what is on the screen with what is in the data. The data may potentially change over time as new news articles are published, so we need to periodically check them and refresh the screen with any new data.



Part 1 - Use the following array to populate a web page which contains news. When the page loads up, it should display all of the news items in the array.

Use an interval function to read the array every 5 seconds. Every time the array is read, remove all news elements from the news container and fill it in with the latest news – so the page is always in sync with the data.

```
let news = [
  { id: 1, title: 'Election Results',
    content: "Newly elected minister..." },
  { id: 2, title: 'Sporting Success',
    content: "World Cup winners..." },
  { id: 3, title: 'Tornado Warning',
    content: "Residents should prepare..." }
];
```

Lab #5: Manage Data 2

This time you will need to add news to the previous array.

When the interval function executes every 5 seconds, it should re-populate the page with all of the articles in the array at that time.

So if a new news item has been submitted via the form and added to the array, the repeating interval will pick it up and include it on the page, together with the previous news items.

Extension: include a button that will stop the interval from reloading the news items.

To update the array, create a form in your page, which will include fields for the title of the news item and the content, and a button to submit this new news item.

There is a trick here. If you use a form and submit, it will trigger a page reload. There are two ways of avoiding this.

1. You can research the prevent default behavior, which stops the form from doing a normal post on its submit event.

   Some prevent default behavior links:

   - W3Schools
   - Mozilla MDN Web Docs

2. You can simply create a form without using an actual html <form> tag. Use text field inputs and a button with onclick event instead.

Data through HTTP

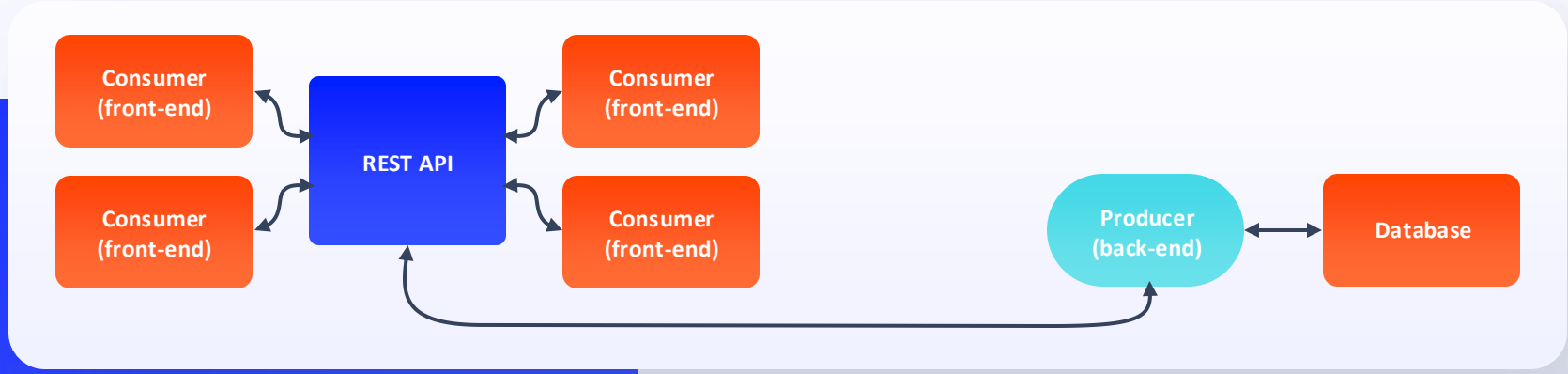
When we talk about data-centric applications, we normally refer to how we manage the data.

We normally use the Producer / Consumer model. A **producer** (**back-end**) will put data into a database and manage it, and a **consumer** (**front-end**) will request data from the database and **consume** it by displaying it to the user.

Generally, we produce once, and consume many times.

This is commonly managed by using HTTP methods to interact with (or consume) a Rest API.

You will learn more about this in module 5, for now, it is enough to understand that data is accessed through a Rest interface using a **Producer** / **Consumer** model.



REST APIs

Our stack is based on RESTful interfaces.

A **REST API** (also known as RESTful API) is an **application programming interface** (API or web API) that satisfies the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for **representational state transfer** and was created by computer scientist Roy Fielding in around 2000.

What makes it RESTful:

- A client-server architecture of clients, servers, and resources, with requests managed through HTTP.
- It is Stateless, with no client information stored or remembered between requests
- Uniform, well-defined interfaces
- Cacheable data that streamlines client-server interactions.

HTTP Methods

HTTP defines a set of **request methods** indicating the desired action to be performed for a given resource. These request methods are sometimes referred to as *HTTP* verbs.

GET is the most common, and is used whenever you browse the web and request data from various websites. In this course, we will mainly focus on the four main methods, but there are more, which offer more complex interactions.

GET

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

POST

The POST method is used to submit a new entity to the specified resource, often causing a change in data stored on the server.

PUT

The PUT method replaces all representations of the specified existing resource with the request payload.

DELETE

The DELETE method deletes the specified resource.

Source : HTTP request methods

HTTP Request Examples

We normally refer to **request methods** through their **resources**. For example, we do not refer to a function for creating a user as “createUser” but it will be a **POST** request on a **/users** resource.

HTTP action /URL		Meaning
`GET /things`	→	Retrieve list of **things**
`GET /things/23`	→	Retrieve one **thing** identified with 23
`POST /things`	→	Create a new **thing**
`PUT /things/23`	→	Replace/update thing 23 (or create **thing** 23)*
`DELETE /things/23`	→	Delete **thing** 23

JSON Placeholder API

In this module, we are not touching real backends yet, but we will do the next best thing. The JSON Placeholder API emulates perfectly how a real server behaves.

- We can simply consume the API from JSON Placeholder to get some dummy data for our front end.
- We will use their "/posts" API which will give us a list (an array) of post objects consisting of:

User	Id	Title	Body

GET API URL for getting posts from JSON Placeholder

```
https://jsonplaceholder.typicode.com/posts?_limit=10
```

POST API URL for creating posts to JSON Placeholder

```
https://jsonplaceholder.typicode.com/posts
```

This ***_limit*** query parameter tells JSON Placeholder to give us only a specific number of posts (10 in this case).

GET Data

We use the inbuilt **fetch** function of the browser to request data. **fetch** makes use of promises via the **then** method.

Because the data is **async**, you are telling the application "when you receive a response, do this." We also need to asynchronously parse the JSON content from the HTTP response before we can **consume** (display or work with) the returned array of posts.

This will display the last 10 posts in the console. You can change the limit - try to put 1, or 100.

```
<html>
   <head>
       <title>JSON Placeholder Posts</title>
   </head>
   <body>
       <h1>Check the console</h1>
   </body>
   <script>
fetch('https://jsonplaceholder.typicode.com/posts?_limit=10')
  .then((response) => response.json())
  .then((json) => console.log(json));
    </script>
</html>
```

POST Data

This time instead of **requesting** or receiving resources, we are **creating** them. Imagine that you are on Facebook and you are posting your latest update.

This is exactly that, you will take the data from the UI and you pass it to the **fetch** function, which will then use the **POST** method to send it to the server. In this case, you are a **producer**.

This time instead, we post data. Post is the action to create a resource. This is not being saved on the server – **it is a fake server after all**.

```
<html>
    <head>
        <title>JSON Placeholder Posts</title>
    </head>
    <body>
        <h1>Check the console</h1>
    </body>
    <script>
 fetch('https://jsonplaceholder.typicode.com/posts', {
   method: 'POST', // GET is default, but we can also use POST
   body: JSON.stringify({ // POSTed data goes in the request body
      title: 'The Studio',
      body: 'Something funny',
      userId: 1,
   }),
   headers: {
      'Content-type': 'application/json; charset=UTF-8',
   },
 })
    .then((response) => response.json())
    .then((json) => console.log(json));
    </script>
</html>
```

Using Axios instead

Fetch is built-in browser functionality. You can further explore industry standards such as the Axios library.

Axios is an http client to call APIs.

- It is a promise-based HTTP client for the browser and Node.js.
- It is used to make XMLHttpRequests from the browser.
- You can use **Axios** to simplify API calls.

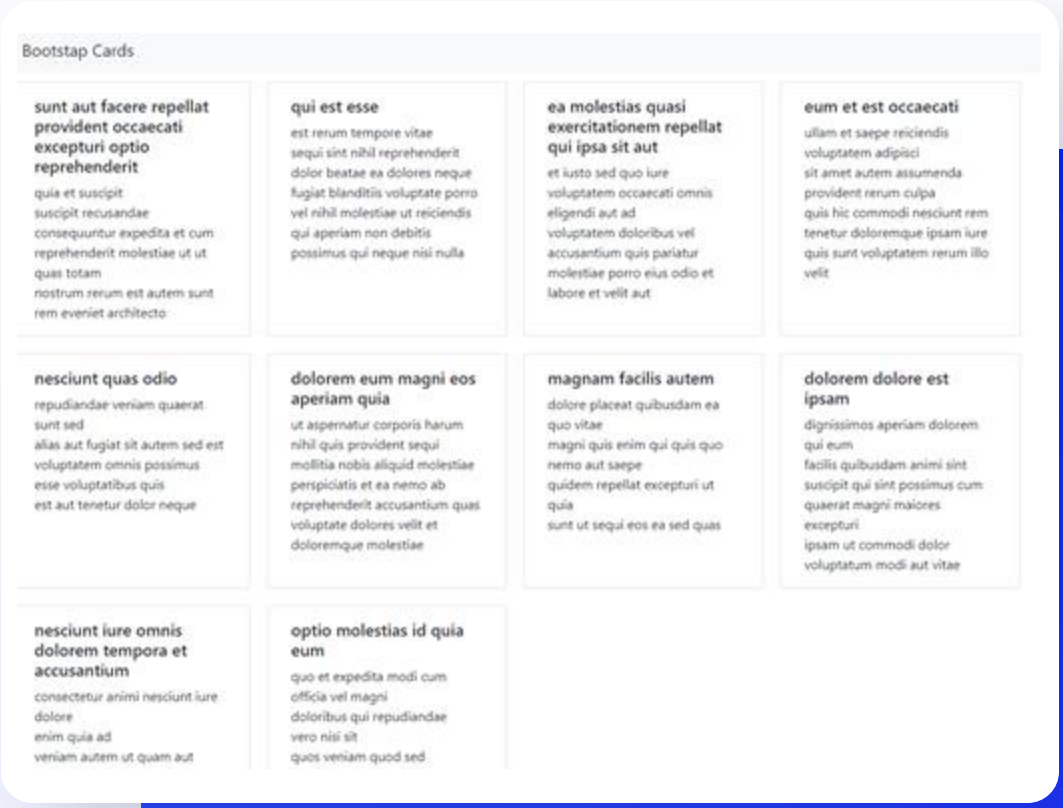
```
<html>
    <head>
        <title>JSON Placeholder Axios</title>
    </head>
    <body>
        <h1>Check the console</h1>
    </body>
    <!-- need to include the axios script first -->
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/axios/1.4.0/axios
.min.js"></script>
    <script>
axios.get('https://jsonplaceholder.typicode.com/posts')
        .then(response => console.log(response.data))
    axios.post('https://jsonplaceholder.typicode.com/posts',
        { // POSTed data goes in the request body
            title: 'The Studio',
            body: 'Something funny',
            userId: 1,
        })
        .then(response => console.log(response.data))

    </script>
</html>
```

Lab #6: Fetch Data

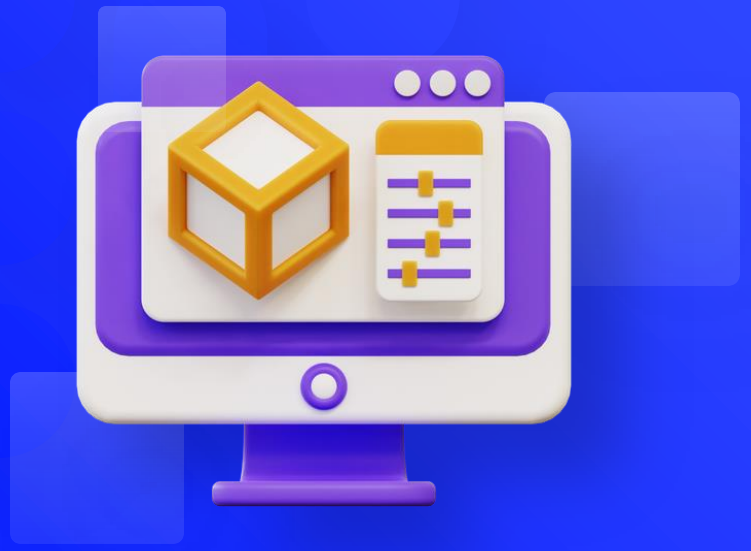
Create a web page that will read the posts from JSON Placeholder using **fetch**.

Using techniques from the previous templating labs, use bootstrap cards and grids to layout the returned post data from JSON Placeholder on the page.

Make use of the fetch API to retrieve data online and display it. Set the limit default to 10. When the page loads up, it will use the default value.



ENHANCE THE APPLICATION



CSS Animations

CSS allows animation of HTML elements without using JavaScript or Flash!

To create a CSS animation, we need to use the **animation** property or its sub-properties, which allows you to configure the animation **time**, **duration** and other animation details.

The actual representation of the animation, is implemented by the @keyframes rule. You may be familiar with the concept of **keyframes** from working with videos, animation or film-making.

Let's start by looking at the animation sub-properties.

Animation sub-properties

- animation-name (The name of the keyframe described by @keyframes.)
- animation-delay (Sets the delay, which is the time between when the element is loaded and when the animation sequence begins.)
- animation-direction (Sets whether the animation will run in reverse after each run, or whether it will return to the start position and run again.)
- animation-duration (Sets the duration of the animation for one cycle.)
- animation-iteration-count (Sets the number of times the animation will repeat, you can specify infinite to repeat the animation.)
- animation-play-state (Allows animation to be paused and resumed.)
- animation-timing-function (Sets the animation speed.)
- animation-fill-mode (Specifies how to apply a style to the target element before and after the animation is executed.)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>CSS Animation</title>
    <style type="text/css">
        p {
            animation-duration: 3s;
        }
    </style>
</head>
<body>
    <p>Hello World!</p>
</body>
</html>
```

We can add animation to the <p> element, but nothing will happen because we didn't add a keyframe here.

Next, we will create an Animation by using keyframes.

Animation keyframes

Once the timing of the animation has been set, it is time to define how the animation will behave.

This is achieved by creating two or more keyframes using **@keyframes**. Each keyframe describes how the animated element should be rendered at a given point in time.

keyframes use a percentage to specify the point at which the animated property change occurs.

0% indicates the first moment of the animation and 100% the final moment of the animation.

Because of the importance of these two points in time, there are special aliases: **from** and **to**, both of which are optional; if from/0% or to/100% is not specified, the browser will calculate the value to start or end the animation.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>CSS Animation</title>
    <style>
        p { animation-name: slidein;
            animation-duration: 3s; }

        @keyframes slidein {
            from { margin-left: 100%; }
            to { margin-left: 0; }
        }
    </style>
</head>
<body>
    <p>Hello World!</p>
</body>
</html>
```

1. We create @keyframes and name it as slidein
2. Then add that animation-name to the style

In this example the <p> element will slide from the right to the left of the browser window.

Animation keyframes

We add a new keyframe.

When the animation is halfway through, we increase the font size and set the distance from the left to 40% to make the text appear to be in the middle.

We also set `animation-iteration-count` to 'infinite' which means it will repeat forever.

We can also change `animation-iteration-count` to a number. If we change it to '3', it will repeat 3 times.

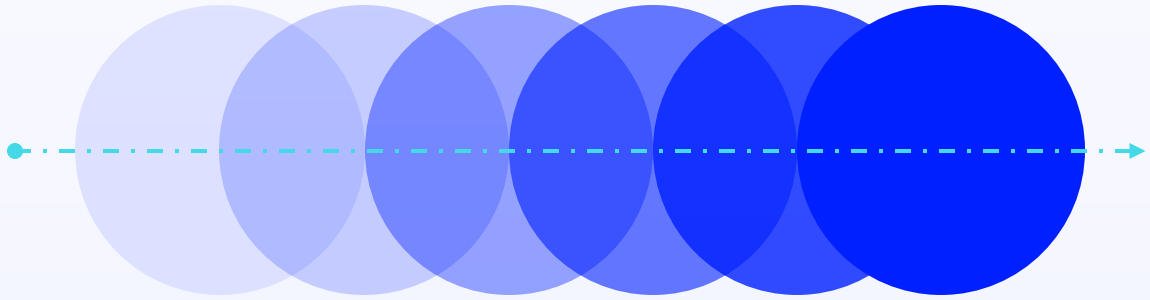
Almost all CSS properties are animatable. **Colour** and **positioning** are commonly animated, as are transformations such as rotation and translation.

> See Creative and Unique CSS Animation Examples for some animation examples and ideas

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>CSS Animation</title>
    <style>
        p { animation-name: slidein;
            animation-duration: 3s;
            animation-iteration-count: infinite; }

        @keyframes slidein {
            from { margin-left: 100%; }
            50% { font-size: 300%; margin-left: 40%; }
            to { margin-left: 0; }
        }
    </style>
</head>
<body>
    <p>Hello World!</p>
</body>
</html>
```

Lab #7: CSS Animation

Make a square slider so that it slides from the top to the bottom and returns to its original position, changing background colour in the process.



Animation Example 1

Animation Example 2

CSS Transitions

CSS animations are controlled directly via the animation properties, and don't directly relate to an action taken by users viewing the page.

We can also use **transitions**, to create a gradual transition between two values of a CSS property. This is most commonly used when **hovering** over an element, using the :hover pseudo-class, to apply a gradual transition between different values on hovered and non-hovered elements.

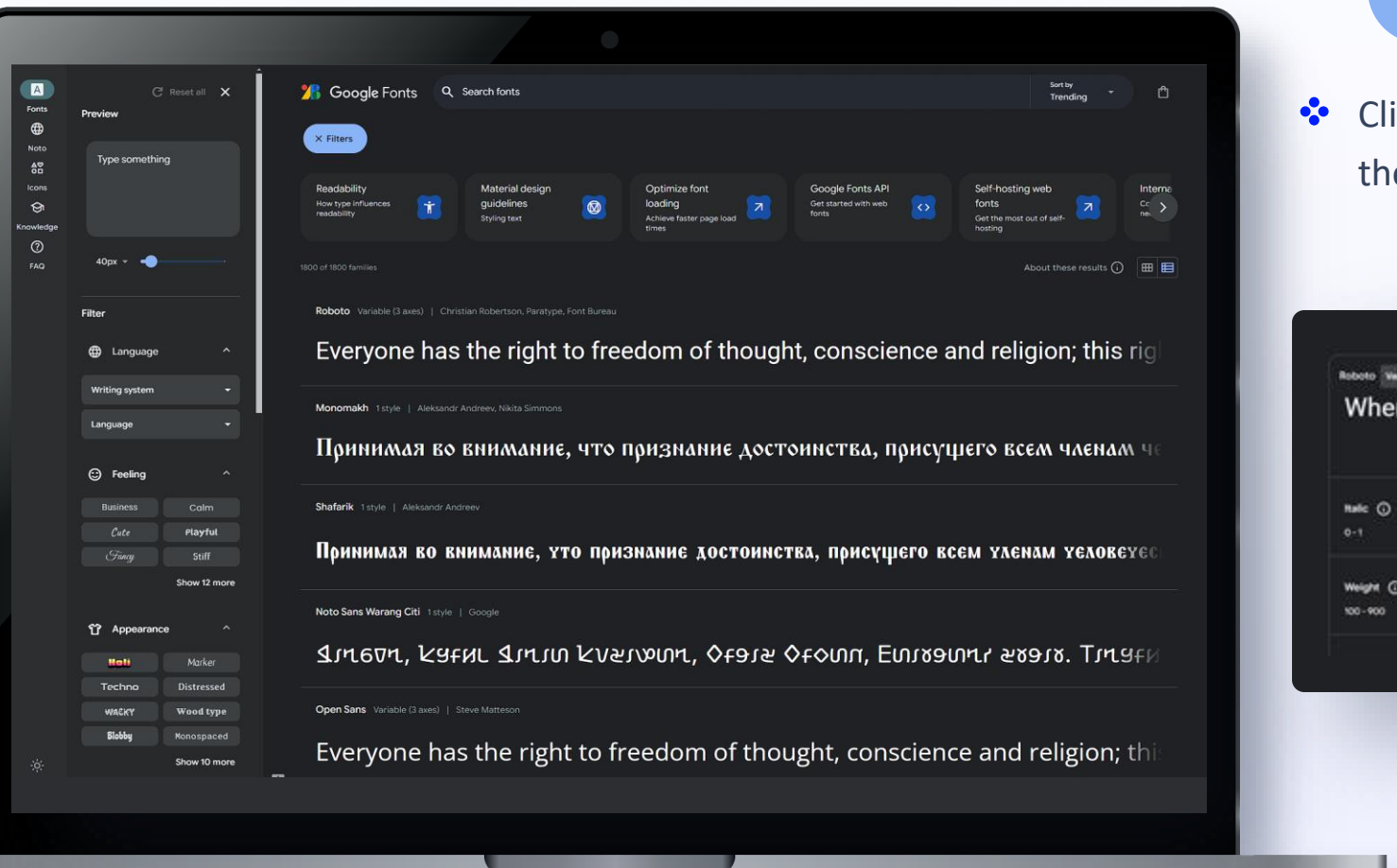
Hovering over the text in this example will gradually change the colour from black to purple.

See Using CSS transitions for more information

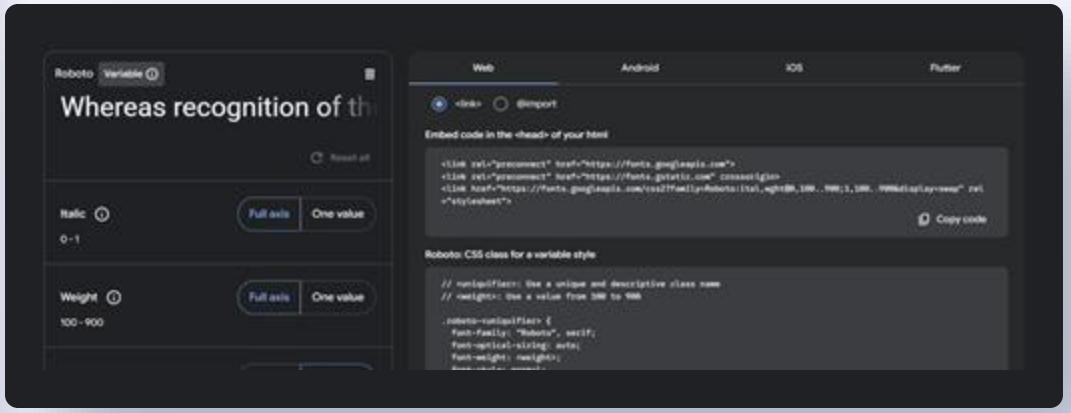
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>CSS Transitions</title>
    <style>
        p { font-family: sans-serif;
            font-size: 24px;
            font-weight: bold;
            colour: black;
            transition: colour 2s; }

        p:hover { colour: purple; }
    </style>
</head>
<body>
    <p>Hello World!</p>
</body>
</html>
```

Font Libraries

- **Google Fonts**: Google Fonts is a font embedding service library which contains a vast library of publicly available fonts. They are optimised and licenced for the web.



- Find the font/s you want to use, by browsing, filtering and using the preview
- Get font Click on the chosen font, then click Get font to use it
- Click Get embed code, then Change styles to copy-paste the Web code for using your chosen font.

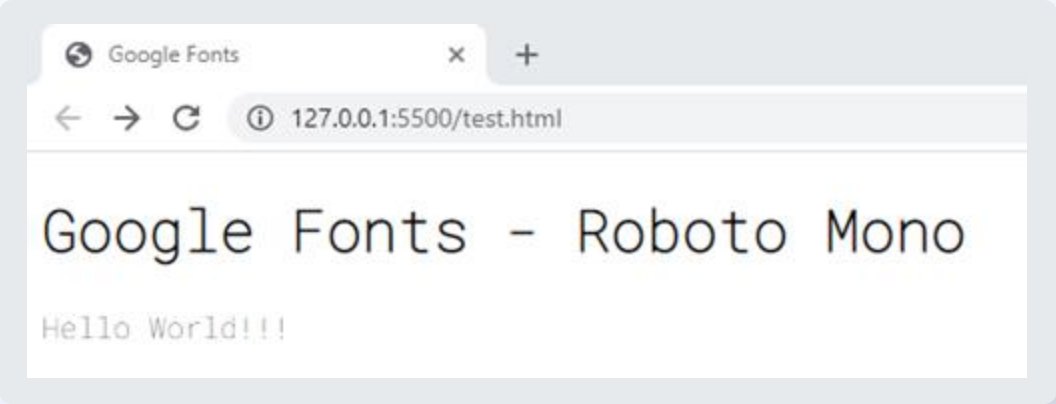


Google Fonts example

The code on the right will allow you to generate this example.

Overall, all you have to do is:

- Reference the font you want to use from Google Fonts.
- Add CSS rules to specify a `font-family`.




```html
<!DOCTYPE html>
<html>
<head>
  <title>Google Fonts</title>
  <!-- Reference Google Font -->
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com"
crossorigin>
  <link
href="https://fonts.googleapis.com/css2?family=Roboto+Mono:wght@100&
display=swap" rel="stylesheet">

<!-- Add CSS rules to specify families -->
  <style>
    body {
      font-family: 'Roboto Mono', monospace;
    }
  </style>
</head>
<body>

<div>
  <h1>Google Fonts - Roboto Mono</h1>
  <p>
    Hello World!!!
  </p>

</body>
</html>
```

Icon Libraries

Iconify.design: Iconify.design is a unified icons framework which consists of over 200,000 vector icons from over 150 different icon sets, all validated, cleaned up, optimized and always up to date.

It is designed to work within the major JS frameworks, as a library in vanilla JS web projects, and even by designers in tools like Figma.

There are many ways to use it which are detailed in the Iconify documentation. We will use their recommended method: **Iconify icon web components.**

This means we just need to register the `iconify-icon` package from their CDN, and then copy-paste the right code for the icons we want to use.

What is a CDN?

A **Content Delivery Network (CDN)** is a system of distributed servers that deliver web content (often universally useful libraries like Bootstrap) to users quickly, based on their auto-detected geographic location.

Why Use a CDN?

- **Faster Loading** : Reduces latency by serving content from servers closer to users.
- **Improved Reliability** : Distributes traffic to prevent server overloads and downtime.
- **Better Performance** : Optimizes asset delivery (e.g., images, scripts, videos).
- **Reliable Versioning** : Provides up-to-date versions of libraries or assets with proper version control.
- **Security Enhancements** : Protects against DDoS attacks and data breaches.

Hosting JavaScript libraries like Iconify on a CDN allows for easy integration and reduced load times. We simply add a single line of code to include the CDN-hosted library and then make use of it.

Icon Libraries

Iconify have their own CDN - make sure to grab the latest version of their script from their website.

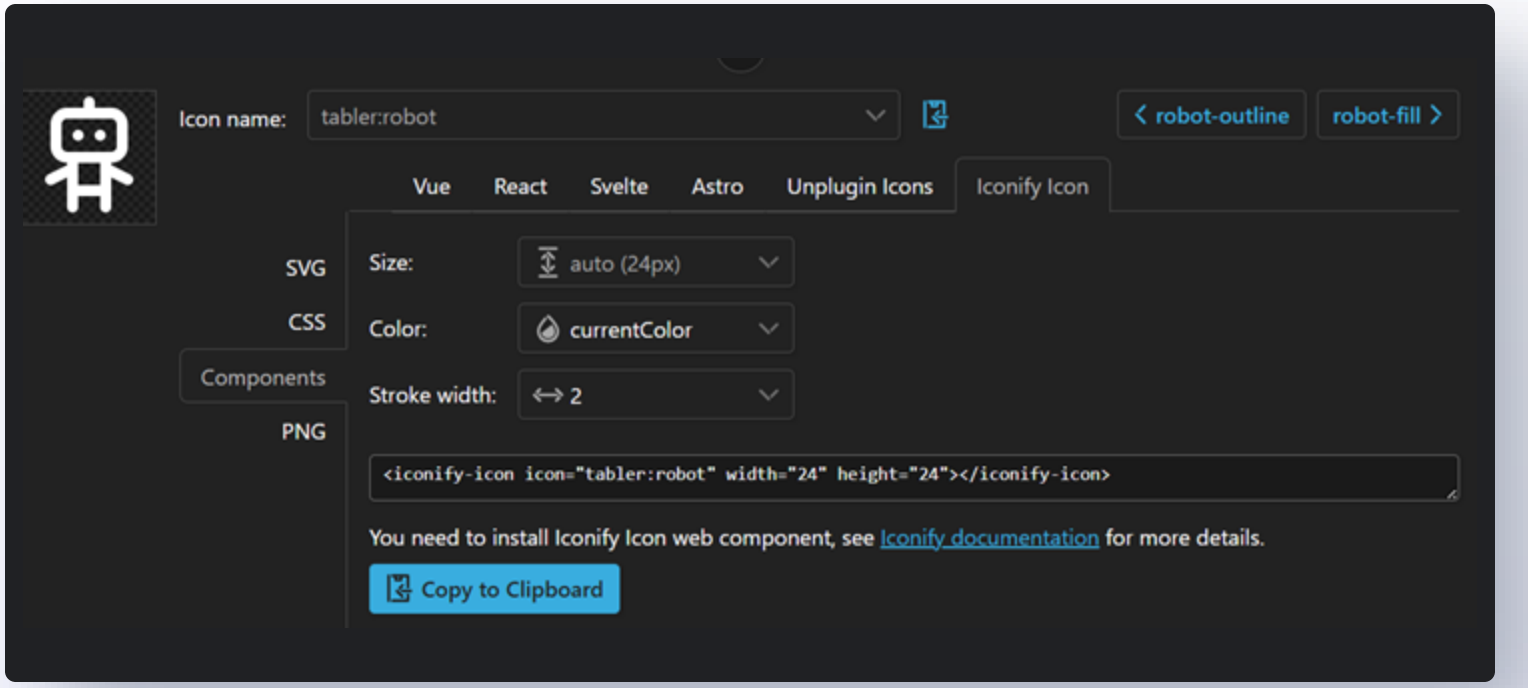
Then go to the list of icon sets and search (top right) for the type of icon you want to use:

```
<script src="https://code.iconify.design/iconify-icon/2.3.0/iconify-icon.min.js">
</script>
```

Icon Libraries

First, select the **Components** **left** tab, and the **Iconify Icon** **top** tab.

Then, customize the **size**, **colour** and stroke **width** as desired.

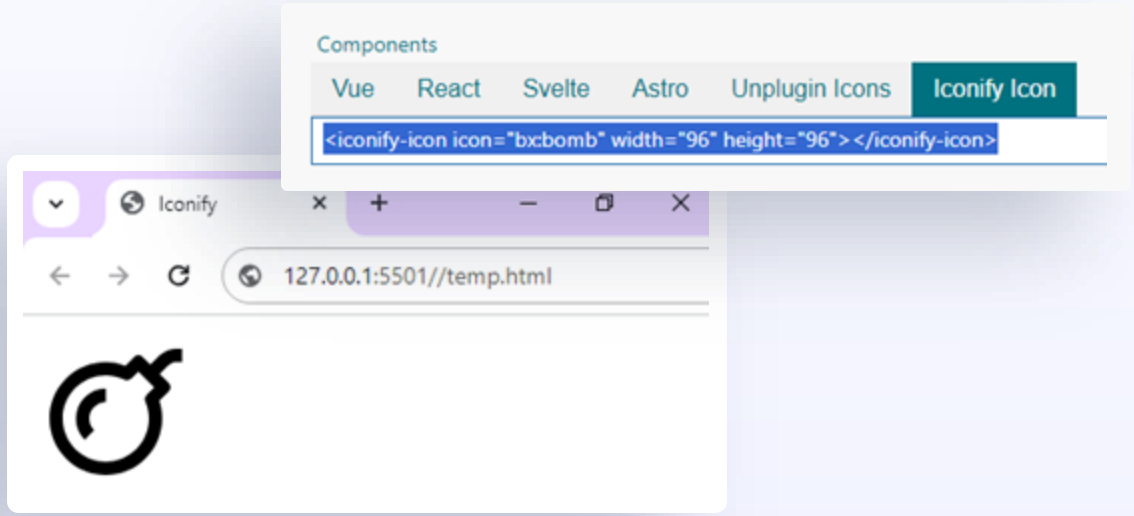


Finally, copy-paste the `<iconify-icon>` code by clicking **Copy to Clipboard**, then pasting into your HTML document. You can further customize (or animate!) the icon using custom CSS.

Iconify Example

The code on the right will allow you to generate this example. Overall, all you have to do is:

- Import the Iconify framework (get the latest version link from here).
- Add copied code of your chosen icon for the Iconify Icon component



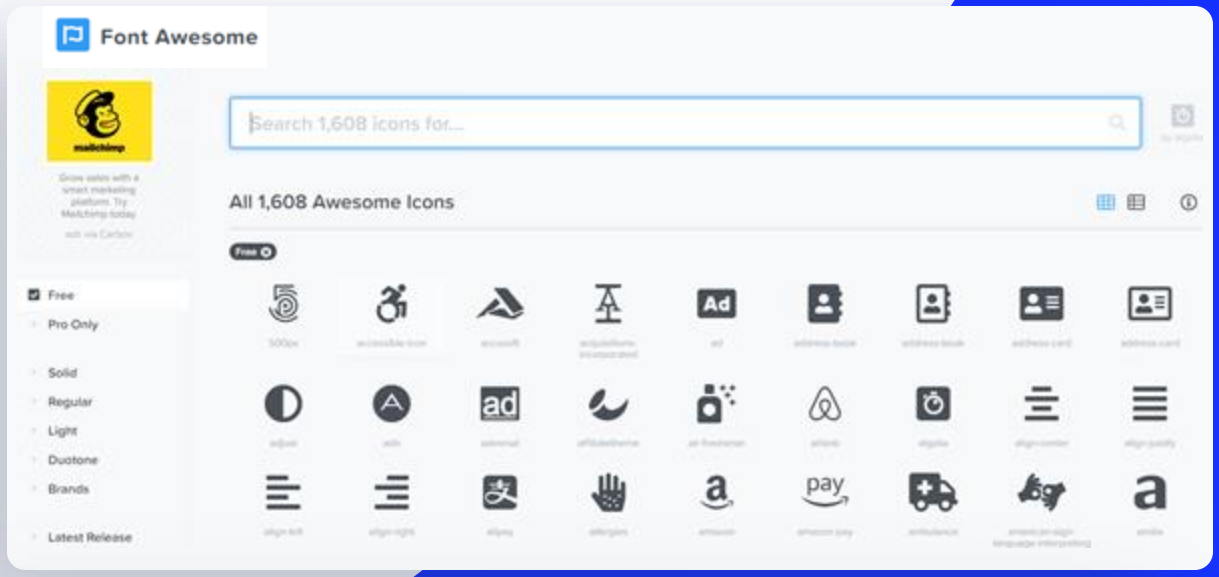

```
<html>
  <head>
    <meta charset="utf-8" />
    <title>Iconify</title>
    <!-- Import Iconify Framework from
https://iconify.design/docs/iconify-icon/#registering-
the-web-component -->
    <script src="https://code.iconify.design/iconify-
icon/2.3.0/iconify-icon.min.js"></script>
  </head>
  <body>
    <!-- Add icon using Iconify Icon component -->
    <iconify-icon icon="bx:bomb" width="96"
height="96"></iconify-icon>
  </body>
</html>
```

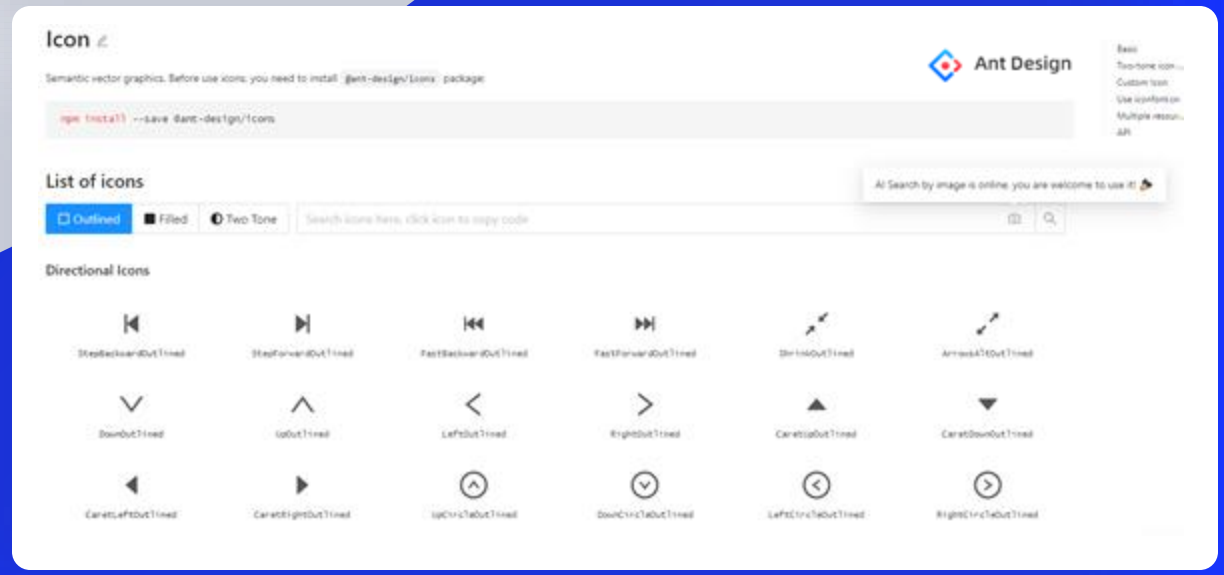
Icon Libraries

Other popular icon libraries:

- Ant.design
- Font Awesome
- Material (Google) icons
- Feather icons

These sets are all included in Iconify.design but can be imported separately as well.





DateTime and Internationalization libraries

- **MomentJS**: Historically probably the most popular JavaScript date library among software engineers. However, over the past few years, modern alternatives have challenged its existence.

DateTime and Internationalization libraries

Moment.js became widely used for its comprehensive and intuitive API for **parsing**, **formatting**, and **manipulating** dates and times, offering excellent support for internationalization and time zones.

It is now considered **legacy** software due to its **large bundle size**, **mutable** data handling, and lack of support for modern JavaScript features like **tree-shaking**, although it still works and is a good choice in some cases.

It can be easily installed or hosted from their suggested CDNs.

Afterwards, simply browse the documentation to find tips and examples of the functionality you need to **parse**, **update**, **manipulate**, **display** or **query** your Moment date/s.

DateTime and Internationalization libraries

Luxon is a powerful, modern, and friendly wrapper for JavaScript dates and times. It supports **DateTimes**, **Durations**, and **Intervals**, with **native time** zone and **Intl** support, using an immutable, chainable, unambiguous API.

Other differences to Moment:

- Months in Luxon are 1-indexed instead of 0-indexed like in Moment and the native JS Date type.
- Luxon uses **getters** instead of **accessor** methods, so `dateTime.year` instead of `dateTime.year()`
- Luxon has both a **Duration** type and an **Interval** type. The Interval type is like Moment's extension library Twix for handling date ranges.

DateTime and Internationalization libraries

Luxon can be easily installed or downloaded from a CDN similar to Moment, by following their basic browser setup.

Their documentation is packed with useful tips on getting started (take the quick tour), as well as comprehensive explanations and examples on all of the core features, and even a cheat sheet for the most common methods.

As with the Moment documentation, it is a simple matter of **knowing** what you want to do with your date, **finding** the relevant section, and **copying** and **adapting** the examples.

Common tasks such as formatting, parsing, comparing & calculating, working across time zones, and checking validity are all supported and explained.

Browse through and try it out!

Luxon Example

The code on the right will demonstrate how to do some basic tasks with Luxon. The main steps are:

- Import the Luxon library.
- Create a new DateTime using the `luxon` object.
- Manipulate the date if/as needed.
- Display the date, formatted as needed, in the right location.

```
<html>
<head>
  <title>Luxon DateTime Formatting</title>
  <script
src="https://cdn.jsdelivr.net/npm/luxon@3.5.0/build/global/luxon.min.js"></script> <!-- reference Luxon library -->
</head>
<body>
<h2>Luxon Date</h2>
<!-- container for Luxon output -->
<div id="displayLuxon"></div>

<script type="text/javascript">
(function() {
  // create a shortcut and a new Luxon date
  const DateTime = luxon.DateTime;
  let nowLuxon = DateTime.now();

  let eDisplayLuxon =
document.getElementById('displayLuxon');
  eDisplayLuxon.innerHTML = `<p>Unformatted date:
${nowLuxon}</p>`;

  let formatted = nowLuxon.toFormat('MMMM dd, yyyy');
  eDisplayLuxon.innerHTML += `<p>Formatted date:
${formatted}</p>`;

  let newYork =
nowLuxon.setZone('America/New_York').toLocaleString(DateTime.
DATETIME_FULL);
  eDisplayLuxon.innerHTML += `<p>New York date:
${newYork}</p>`;
})();
</script>
</body>
</html>
```

Lab #8: Luxon Dates

Use the Luxon documentation to help in answering these exercises.

***Extension**: Try re-doing the exercise using the Moment.js library instead.*

Using the Luxon library, try to solve the below problems:

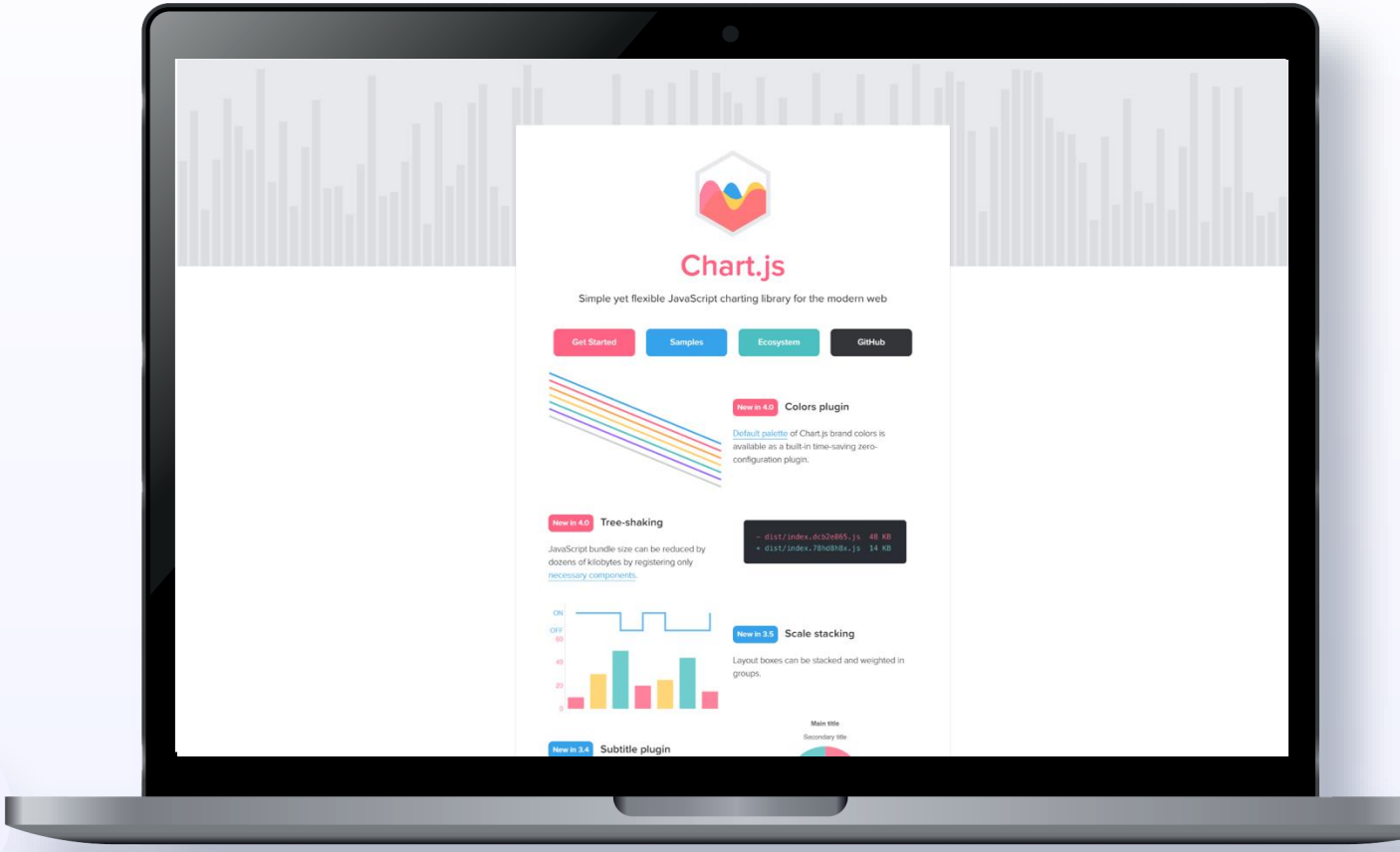
1) Calculate the number of days between your birthdate and the current date
2) Display the number of years, months, and days between your birthdate and current date

Example: 24 years, 8 months, and 26 days

3) Given two dates, display the date closest to the current date
4) Given two dates, display whether the first date is before or after the second date
5) Display the current time in London

Charts libraries - Chart.js

Chart.js: A popular, beginner-friendly, lightweight JS charting library. Generates simple, easy-to-implement charts for basic visualizations. See more: chartjs.org



Charts libraries - Chart.js

Strengths:

- Easy to use and quick to set up for basic charts.
- Lightweight and efficient for small to medium datasets.
- Ideal for simpler, visually appealing charts.

Weaknesses:

- Lacks support for advanced or highly customised charts.
- Limited scalability for large datasets or complex visualizations.
- Fewer built-in interactivity and customisation options compared to other libraries.

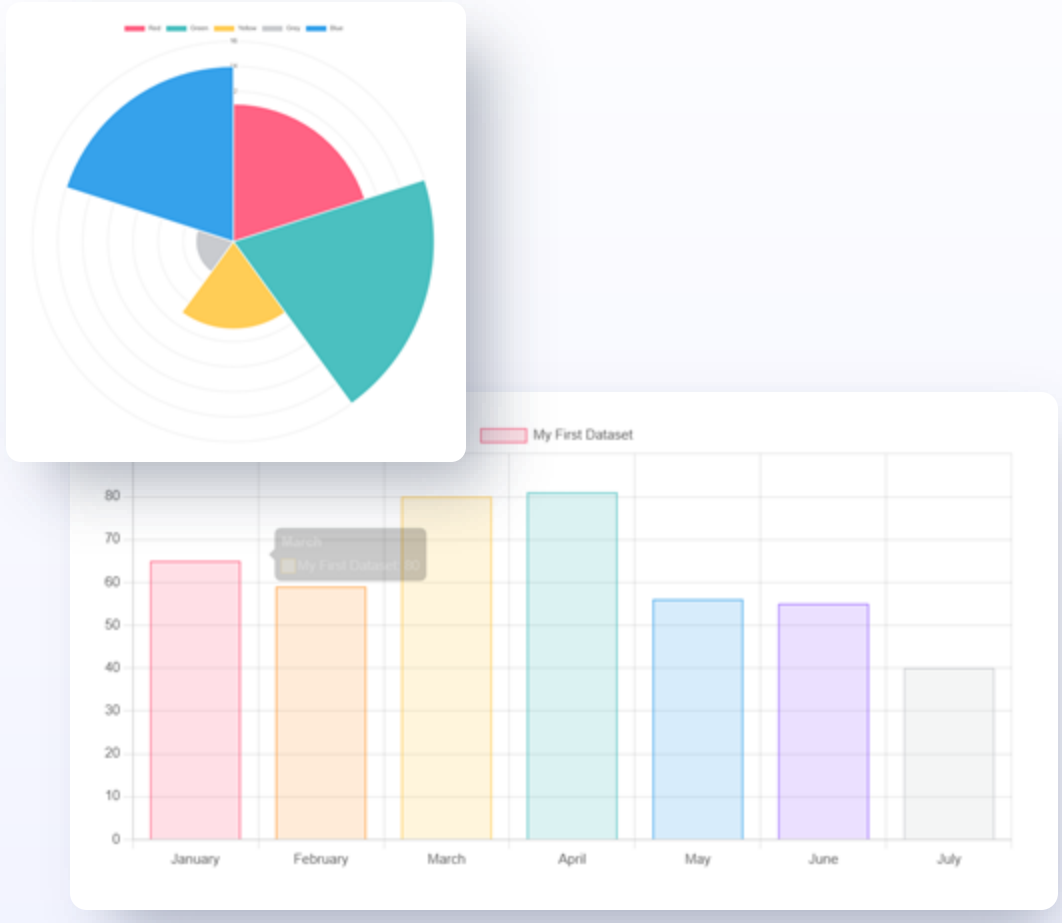


Chart libraries - eCharts

Apache Echarts: is a huge open-source JavaScript library for web-based data visualization developed by Apache.

See more: echarts.apache.org

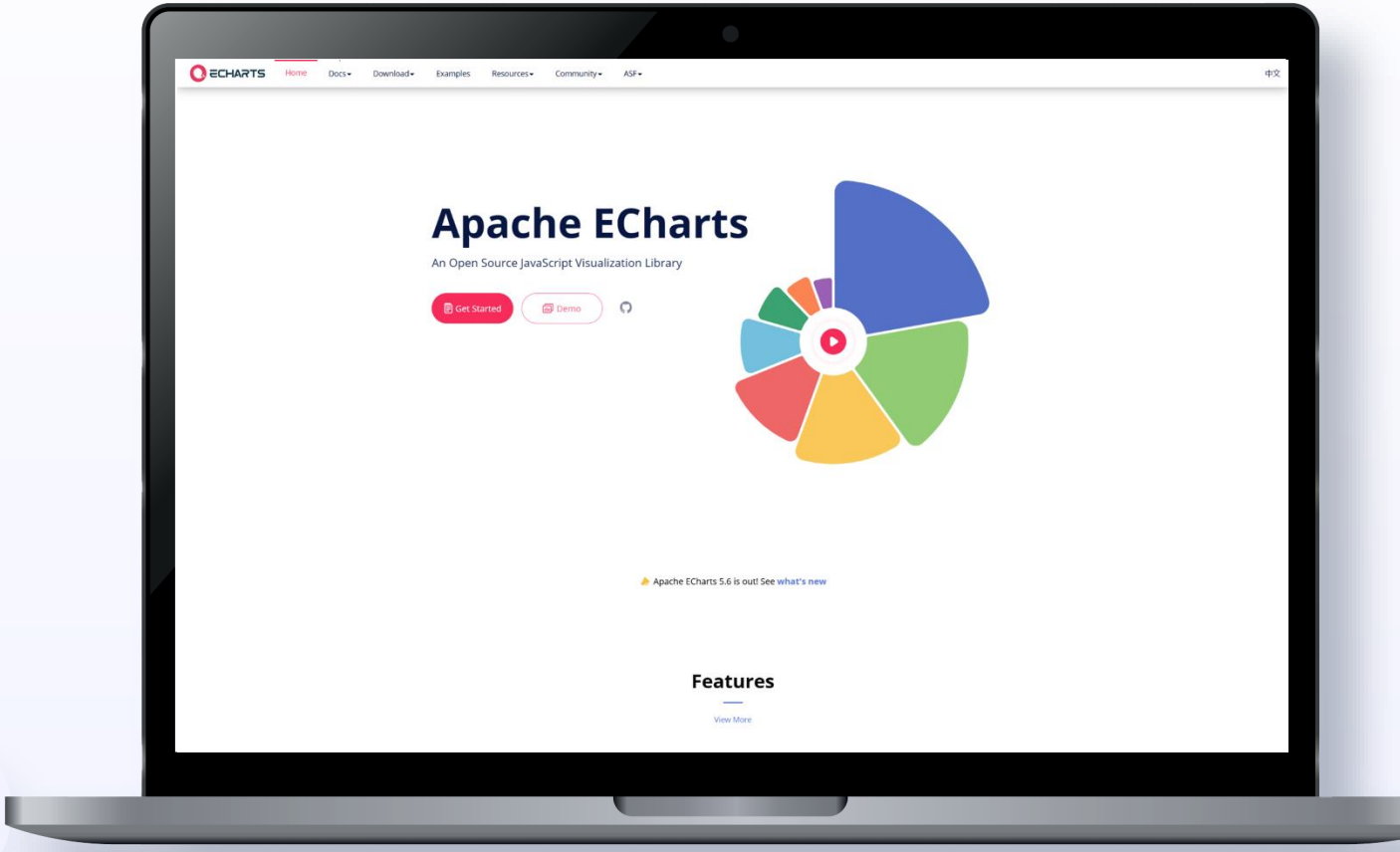


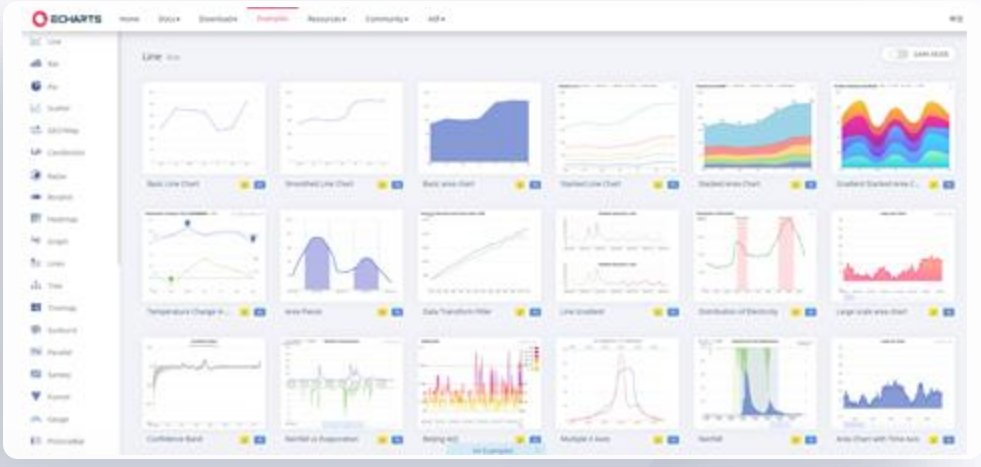
Chart libraries - eCharts

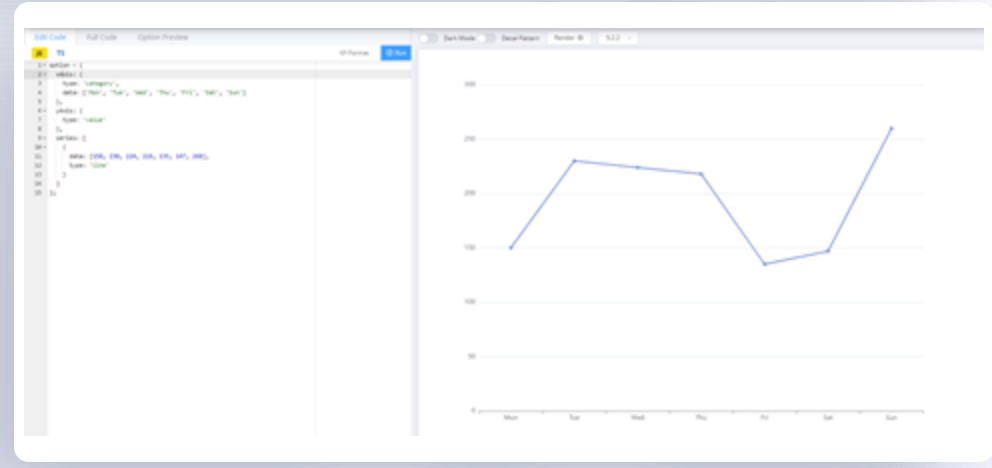
Strengths:

- High flexibility and support for complex visualizations (e.g., 3D, geographic maps).
- Better suited for enterprise-level or data-heavy projects.
- Excellent for interactive and dynamic charts.

Weaknesses:

- More complex to learn and implement.
- Larger file size and can be more resource-intensive.
- May not be necessary for simple or smaller projects.

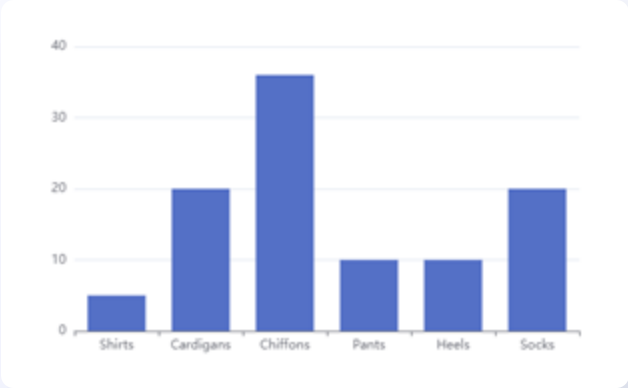




eChart Example

The code on the right will allow you to generate this example. Follow the Get Started guide to:

- Select a to place the chart
- Create a chart object using echarts.init
- Pass the option data to the chart object and render it using the setOption method



Try adding another bar to the chart and another series to the data!

```
<html>
  <head>
    <meta charset="utf-8" />
    <title>ECharts</title>
    <!-- Include the ECharts file you just downloaded -->
    <script src="echarts.js"></script>
  </head>
  <body>
    <!-- Prepare a DOM with a defined width and height for ECharts -->
    <div id="main" style="width: 600px;height:400px;"></div>
    <script type="text/javascript">
      // Initialize the echarts instance based on the prepared dom
      let myChart = echarts.init(document.getElementById('main'));

      // Specify the configuration items and data for the chart
      let option = {
        title: {
          text: 'ECharts Getting Started Example'
        },
        tooltip: {},
        legend: {
          data: ['sales']
        },
        xAxis: {
          data: ['Shirts', 'Cardigans', 'Chiffons', 'Pants', 'Heels',
'Socks']
        },
        yAxis: {},
        series: [
          {
            name: 'sales',
            type: 'bar',
            data: [5, 20, 36, 10, 10, 20]
          }
        ]
      };
      // Display the chart using the above configuration items and data
      myChart.setOption(option);
    </script>
  </body>
</html>
```

Lab #9: eCharts

Adapt the code on the right to make a chart that shows how many items are listed under each category on the Fake Store API.

- Get data from the FakeStore API (Fetch or Axios the data)
- Have an eChart bar for each category
- Show how many items are listed under each category

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>eCharts</title>
    <script
src="https://cdn.jsdelivr.net/npm/echarts@5.4.1/dist/echarts.min.js"></script>
</head>
<body>
    <!-- Prepare a DOM with a defined width and height for ECharts -->
    <div id="main" style="width: 900px;height:600px;"></div>

    <script type="text/javascript">

        // Specify the configuration items and data for the chart
        let options = {
            title: { text: 'Fake Store Categories' },
            xAxis: {
                data: ['Category 1', 'Category 2', 'Category 3', 'Category 4']
            },
            yAxis: {},
            series: [{
                name: '# products',
                type: 'bar',
                data: [0, 1, 5, 2]
            }]
        };

        fetch('https://fakestoreapi.com/products')
            .then((response) => response.json())
            .then((json) => {
                console.log(json)
                // use this JSON to find and set correct option data for the chart
            })
            .then(() => {
                // Display the chart
                myChart.setOption(options);
            })

        // Initialize the echarts instance based on the prepared div
        let myChart = echarts.init(document.getElementById('main'));

    </script>
</body>
</html>
```

Lab #10: Create Fake E Commerce Website

Create a fake eCommerce website with the data from the Fake Store API.

- Fetch or Axios the API data
- Create Bootstrap cards to display the items, including the image, title, price, item description
- Create a drop-down selection to choose individual product categories for filtering
- (Optional extra 1) Create a search for item feature
- (Optional extra 2) Add custom icons to each card (To show item category)
- (Optional extra 3) Include a sorting feature to sort items by price or title