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?

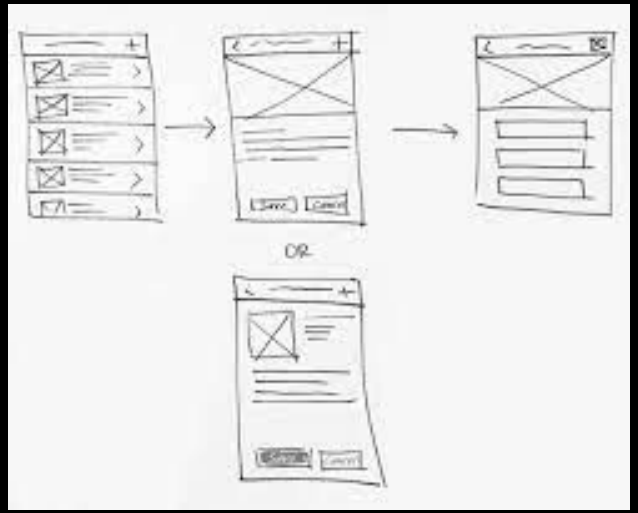
In the IT industry we have many types of designs, here we will focus only on a few:

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

Sketches

Pen and paper, or pencil. Or an iPad, whatever you like, but start drawing it out. The design for our previous application should not take you more than 5 minutes to sketch.

You need to start thinking professionally, **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. Pretty pictures don't do that.

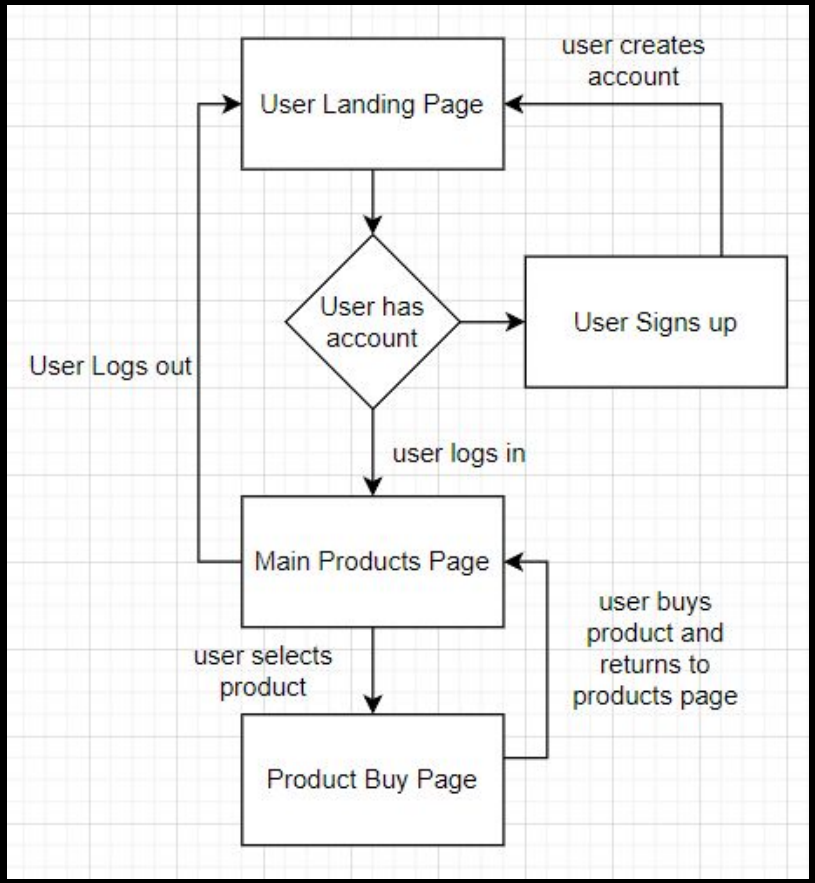


Application Design

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 is very hard not to put your hands on the actual code, but we are still very far away.



Lo-Fi Designs

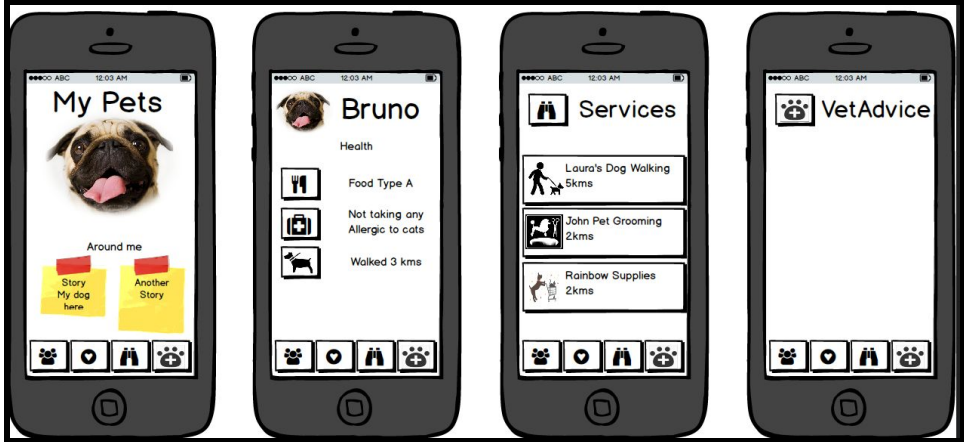
Lo-Fi is the key to acceptance, and also shows that you are professionally trained. Lo-Fi should be as Lo as possible.

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.



What are frameworks in Javascript?

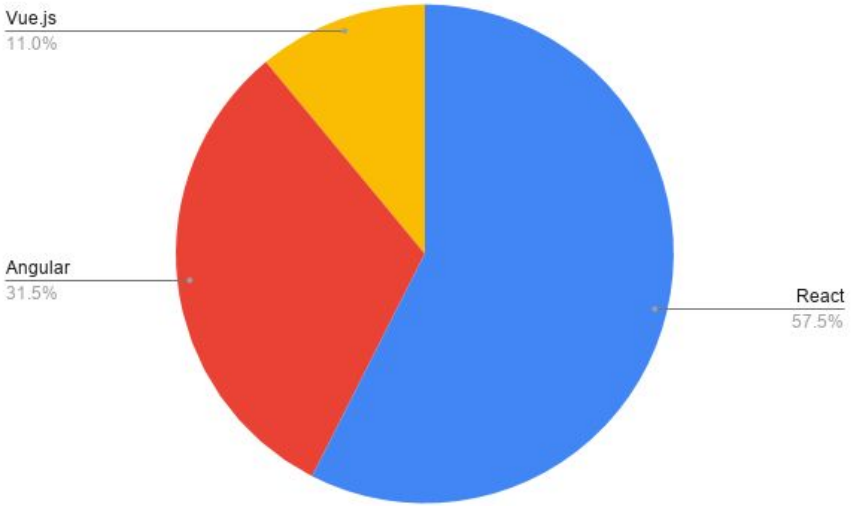
Frameworks can be defined as “scaffolding” for larger applications. Javascript is a “badly” designed language, remember, it was developed in two weeks. In order to overcome this, a number of encapsulating frameworks have been created to “abstract” many modern functionalities and make it much more robust and simpler to develop. For example, there is not a simple way to “update” the UI dynamically. Frameworks like React solve this problem. We normally refer as Vanilla, an application which uses no framework.

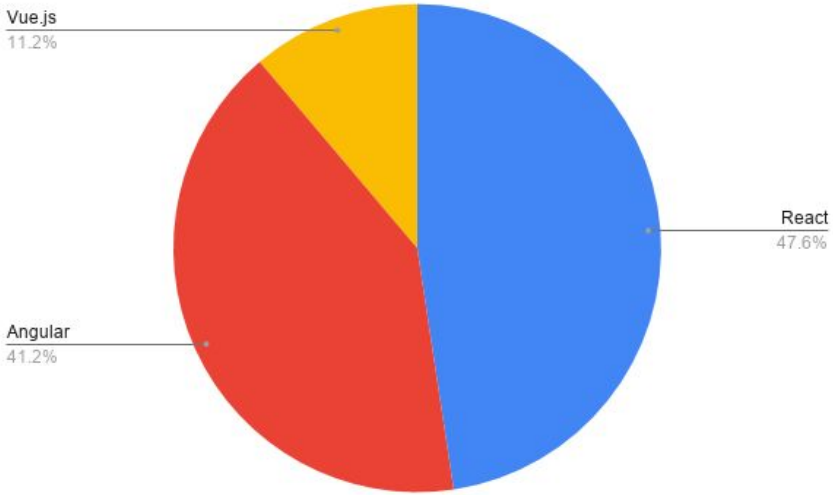
Today the most popular frameworks are React, Angular and Vue, but many others exist, with pros and cons. React is a UI library, Angular is a fully-fledged front-end framework, while Vue.js is a progressive framework. In this program we will focus on React.

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.





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 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 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.

Angular Continued

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 - Meteor / Vue / Next

Meteor.js : Used to develop web and mobile apps, other JS frameworks (like React, Angular) can be embedded as well. Mobile Applications are made possible using Apache Cordova.

Vue.js : An increasingly popular component-based, progressive JS framework for building web-based UIs.

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.

Style Guides

Style Guides are distinct from design pattern libraries, which are a long-standing tool used by UX practitioners to define broad design ideas, rather than specific implementation details.

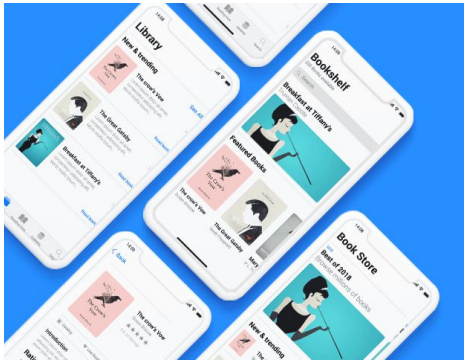
You may wonder “why so many applications all look the same?” That is because we make use of style guides. Almost every major company has a well defined and DOCUMENTED style guide.

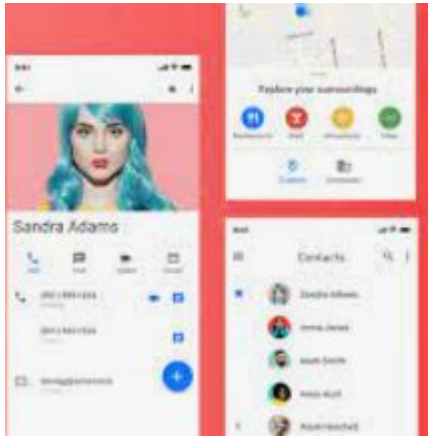
Google has **Material**, Microsoft has **Fluent**, IBM has **Carbon**, Apple has IOS style guides (because Apple is Apple). In this module we will use “Bootstrap”, because it is a widely used, more generic styling framework, and it will help you to easily transition everywhere.

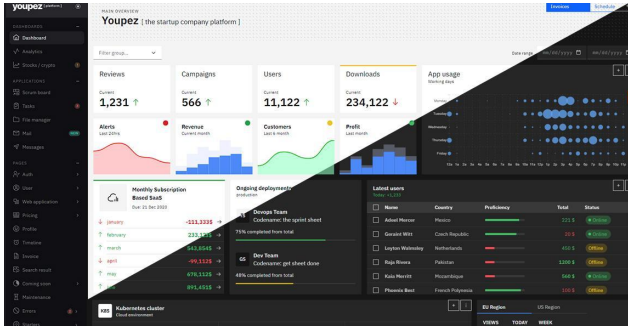
Class Exercise: Some examples

Can you tell which company they belong to?

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







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 favored for quick low-fidelity prototyping. UX teams spend time away from computers, using sticky notes, whiteboards, and notepads to annotate paper prototypes.

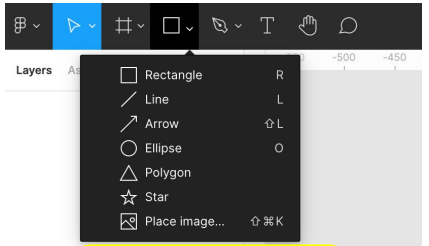
Effective planning lets designers swiftly create wireframes, mockups, and prototypes after this preliminary phase.
Paper prototyping is crucial for early UX design as it encourages collaboration and cost-effective exploration of various ideas.

What is Figma / Why it has become popular

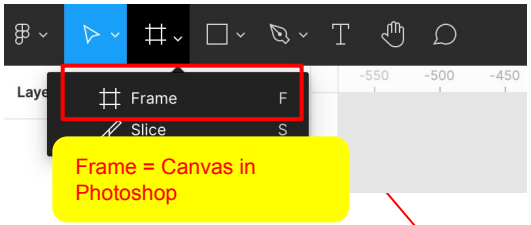
- Figma is a design / prototyping tool that can be easily shared with multiple team members and collaboratively work together.
- Note/Feedback from collaborators helps the designer /design teams to get feedback easily.
- Figma can provide developer's code for UI (eg. fonts, colours, sizes)
- Figma is a cloud-based design/prototyping tool which allows the designers to work on their files, both 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 recognised in the designers community as well.

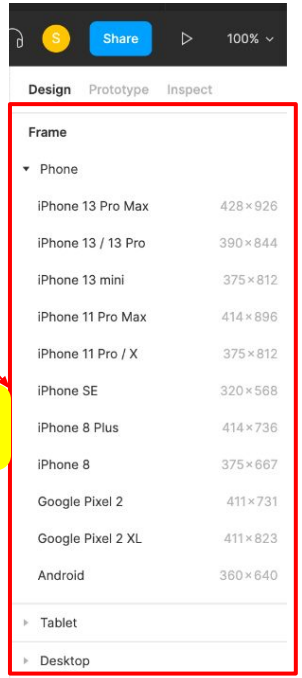
Figma editing tools



Low/Mid fi prototype is basically created with Rectangles/Circles just to show the concept & ideas

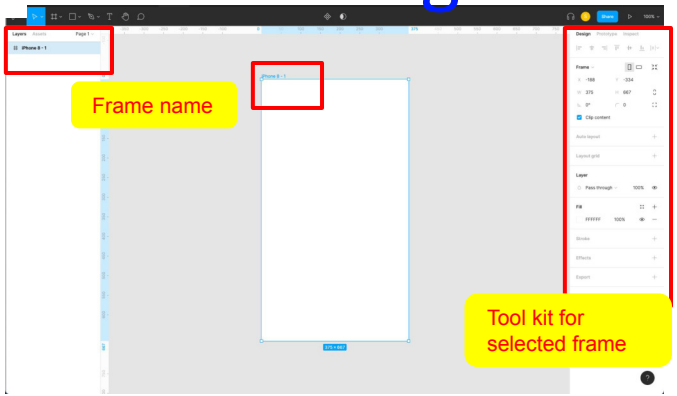


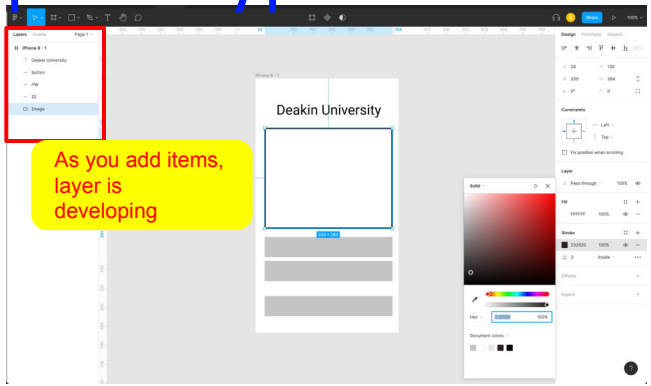
Frame = Canvas in Photoshop

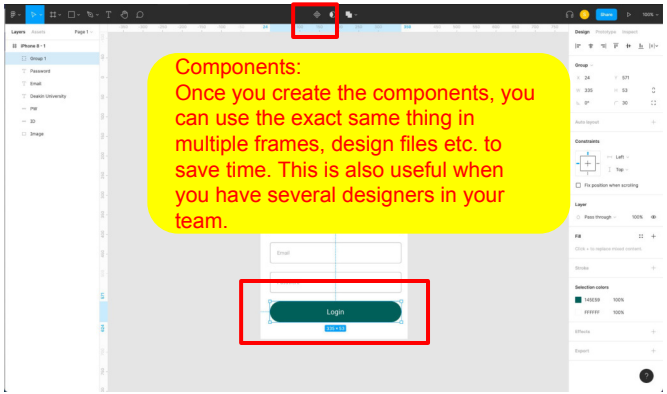


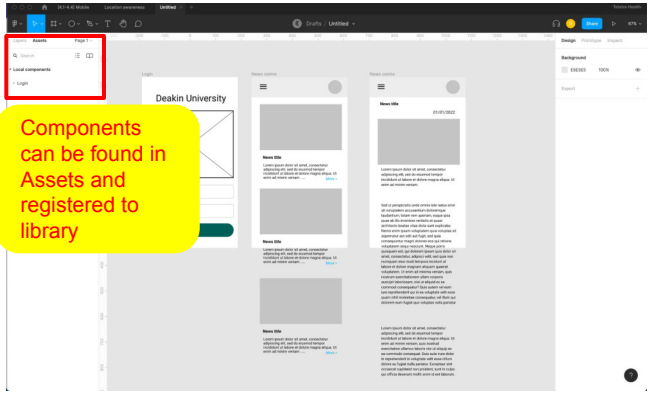
Can choose suitable frames or create your own

Creating low-fi prototype

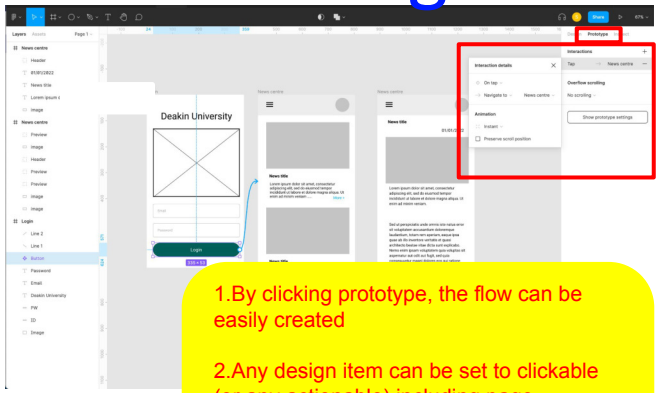








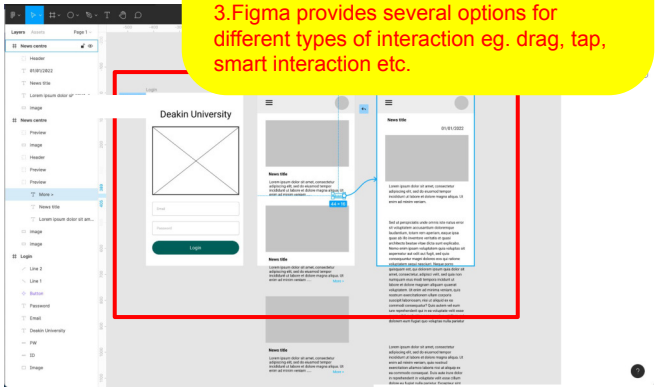
Creating interactive prototypes

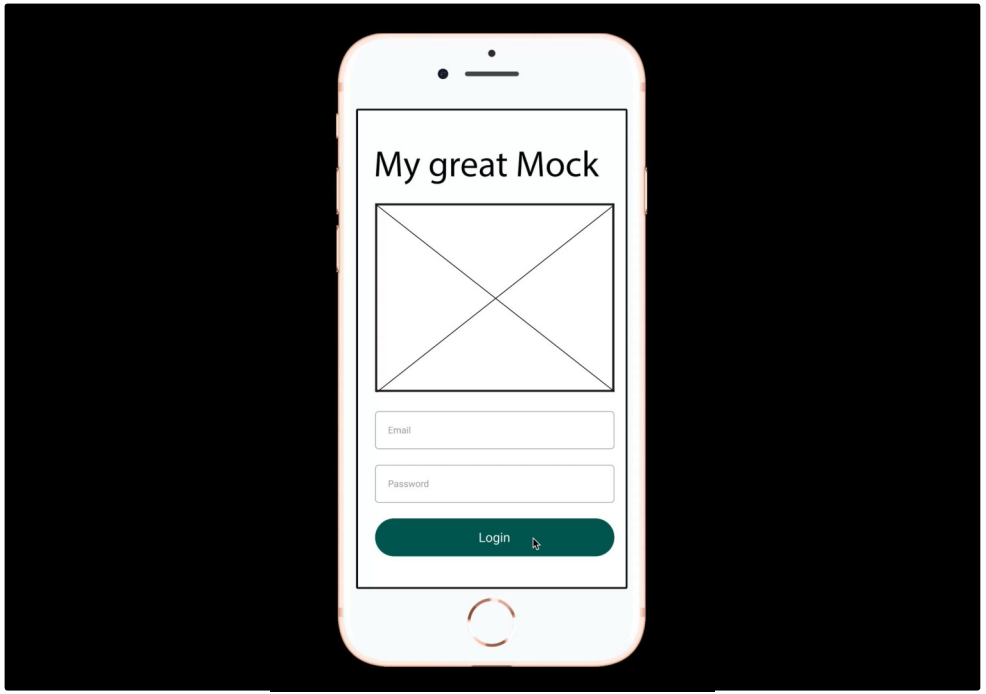


1.By clicking prototype, the flow can be easily created

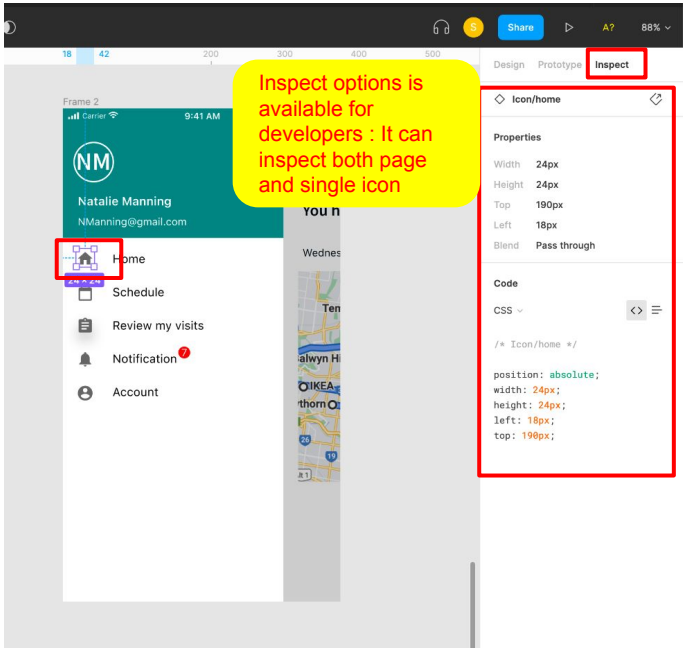
2.Any design item can be set to clickable (or any actionable) including page

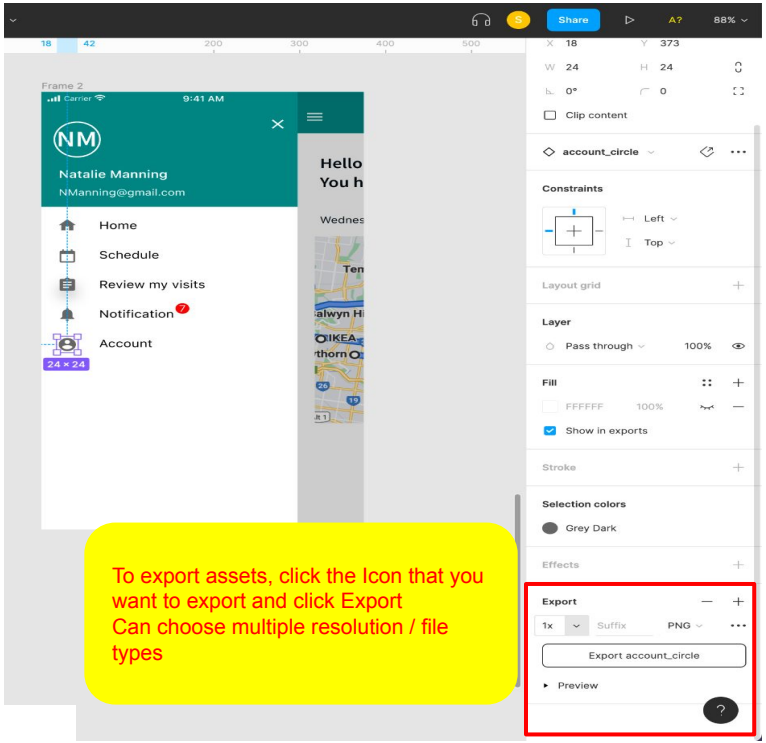
3.Figma provides several options for different types of interaction eg. drag, tap, smart interaction etc.



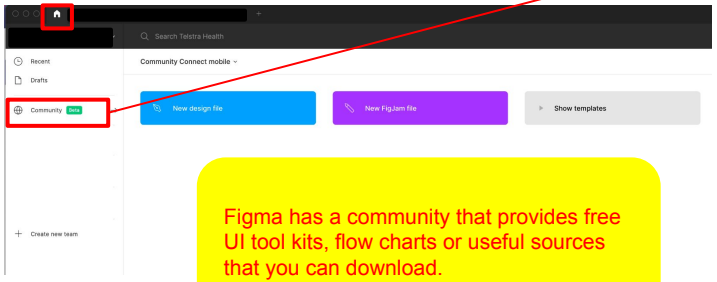


Figma editing tools

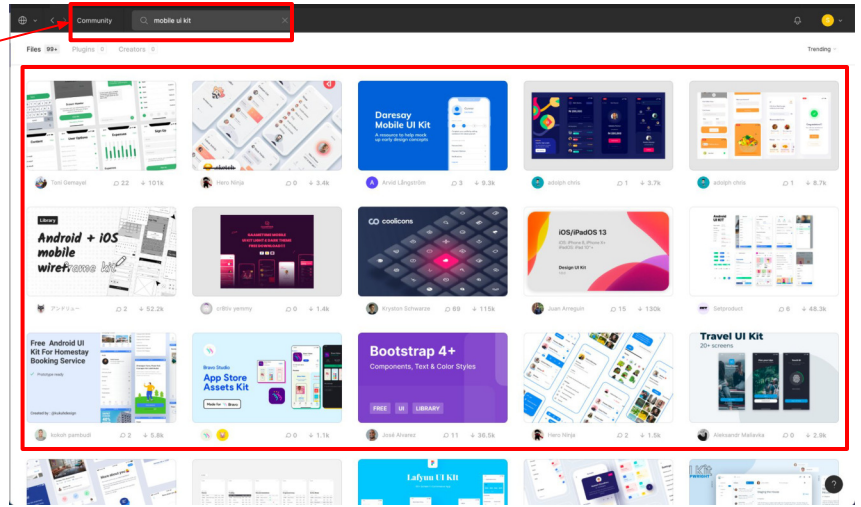




Free UI tool kit or useful source from community



Figma has a community that provides free UI tool kits, flow charts or useful sources that you can download.



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.

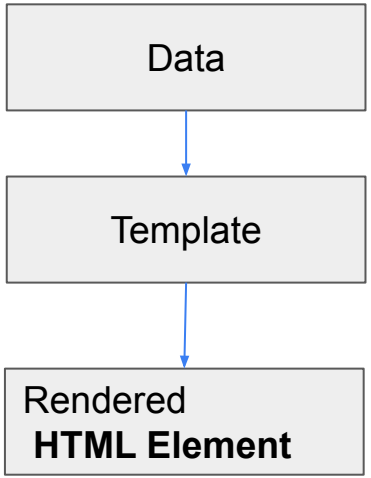
Also, this could be part of your portfolio, so try to do some research and come up with a personal design. For example, it could be a social media application dedicated to the Warhammer Community, or the Ferrari Lovers.

Templating

In the last 20 years, web development has become much more data-oriented. For this reason, we have moved towards the MVC model (Model View Controller).

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.**

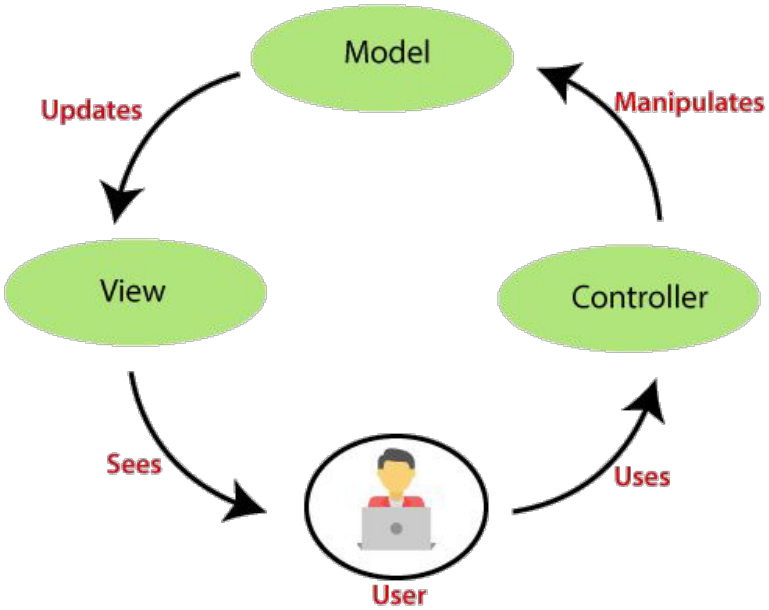
Templates are reusable pieces of code that are normally data-centric.



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.



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 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 **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 alongside the CSS required for the template:

You can do this using more than one template. You can use one single 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 :

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

```
<!DOCTYPE html>
<html>
    <head>
        <style>
            body {
              font-family: Arial, Helvetica, sans-serif;
              background-color: #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.

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

Example of using a template:

We can use the below script to add a card to a HTML page using templates.

```
<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 #2 - 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-im
age"},
        {title: "sky", url:
"https://mymodernmet.com/wp/wp-content/up
loads/2020/11/White-house-night-van-goh-w
orldwide-2.jpg" },
    ]
}
```

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, capture how each function relates, and describe what it will look like.

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 on paper, trying to understand how everything communicates, what causes something else to trigger. For example, buying a product needs to automatically update the stock list, or if you have a login, you should have signup.

3) Define the data, everything spins around data. Define your data models and find any edge cases.

4) Use any tool you prefer to create a prototype, and present it to someone who is not you to test, testing your own designs is a “terrible” practice.

5) Find ideas and define a color scheme

6) Create the HI-FI designs.

Lab #3 - 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

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:

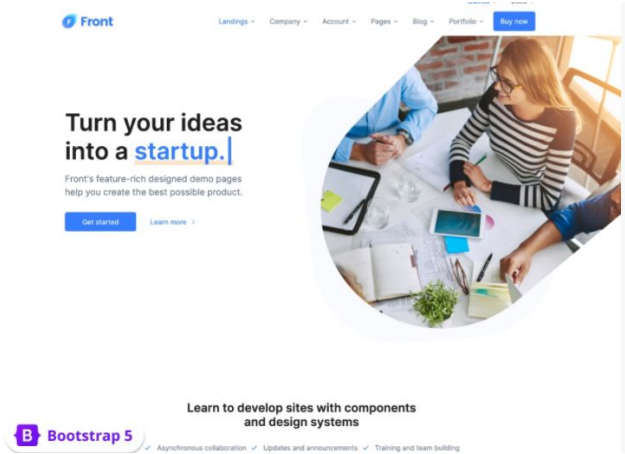
Basic structure with a grid system, link styles, and backgrounds etc.

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 customise 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+VKNBQNGCHeKRQ
N+PtmoHDEXuppvnDJzQIu9" 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-HwwvtgBNo3bZJJLYd8oVXjrBZt8cqVSpeBNS5n7C8I
VInixGAoxmnlMuBnhbgrkm"
        crossorigin="anonymous"></script>
</body>

</html>
```

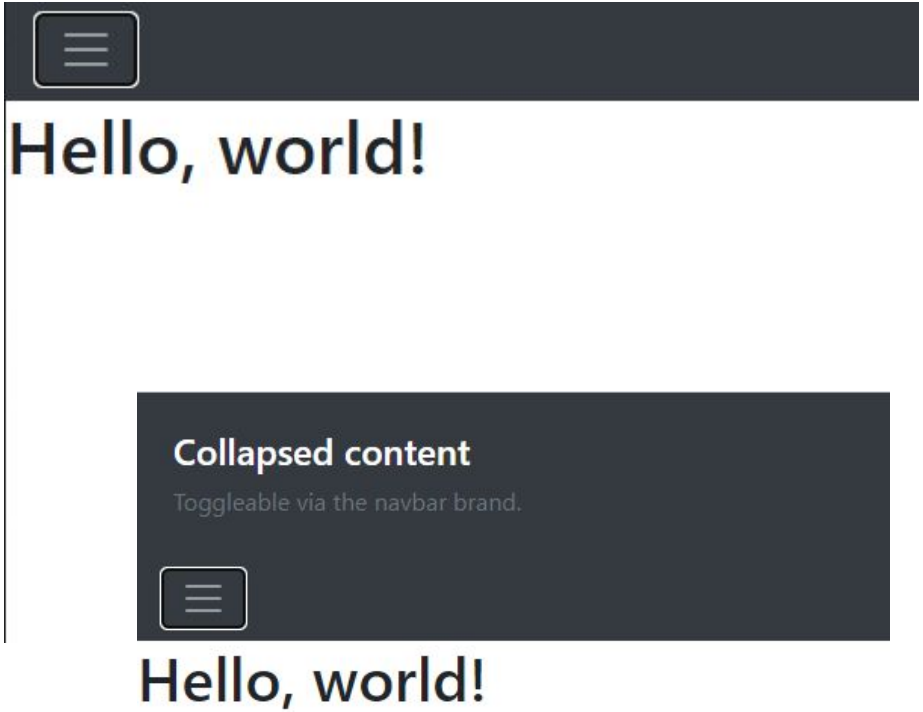
Simple but effective

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.

For example:

```
<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 customise 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)

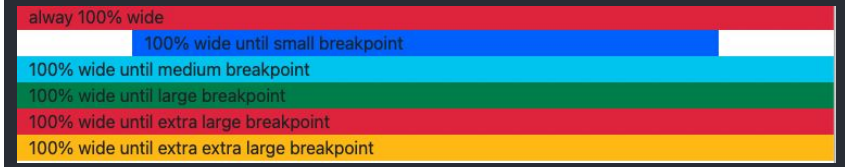
Xx-large(*-xxl) : 1400px and up (large screens)

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>
```

When we open the browser as medium size, it will display:



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 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 customising 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

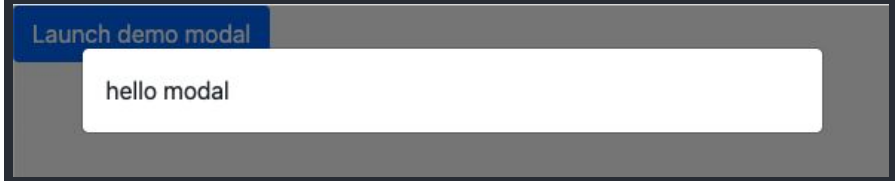
Buttons, Card, Carousel, Dropdowns, Forms, Input Group, List Group, NavBar, Modal, Navs and tabs, Pagination.

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>
```



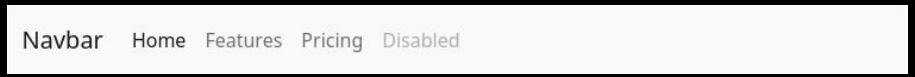
Common Components

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.

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>
```



See Navbar

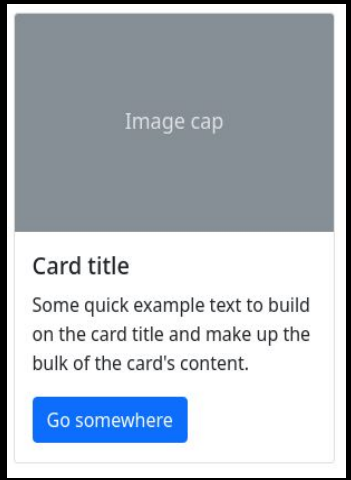
Common Components

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 as cards.

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>
```

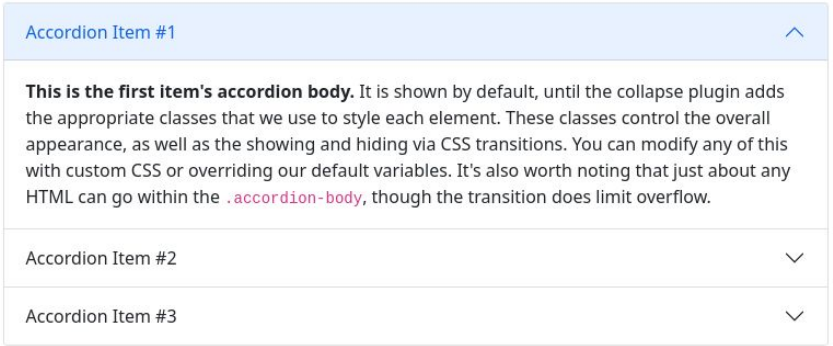


See Card

Common Components

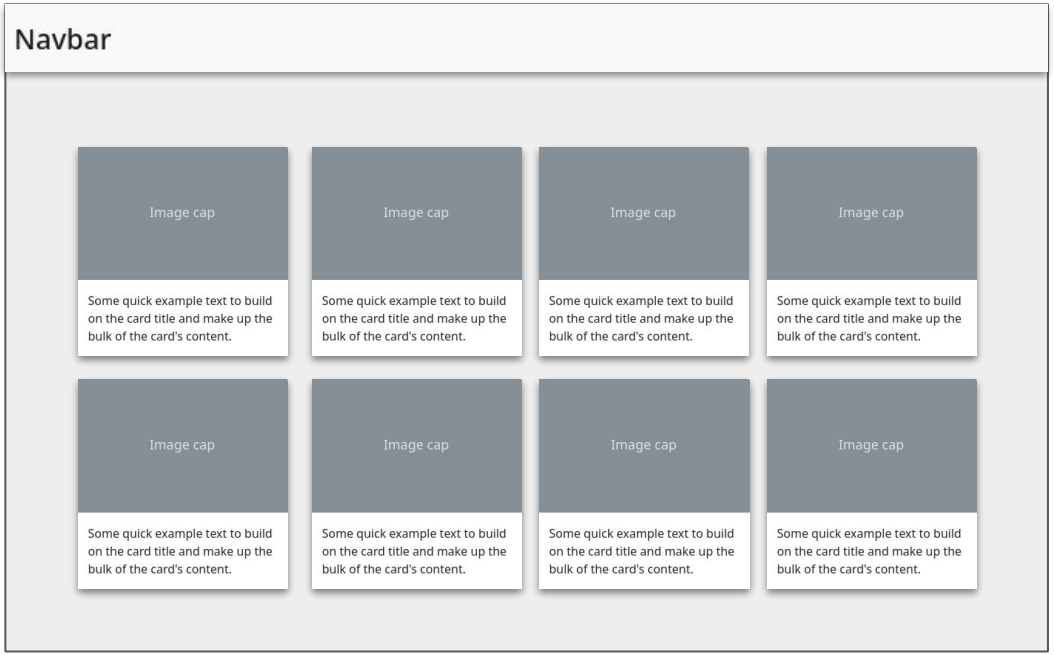
As data becomes more complex, you will need to find better ways to present it to the users.

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



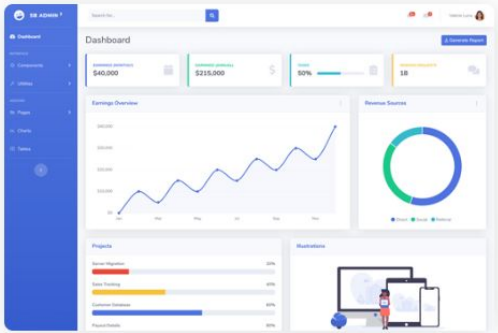
Using only bootstrap elements, create this grid-based card layout, making it responsive.

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.

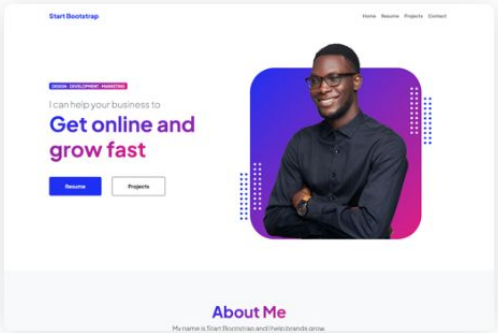
Bootstrap Themes

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

Dynamic Data

We live in a data-centric world, and everything can be represented as data. On the web, we 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 very 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 solving 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* behaviour links:

1. W3Schools
2. Mozilla MDN Web Docs

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

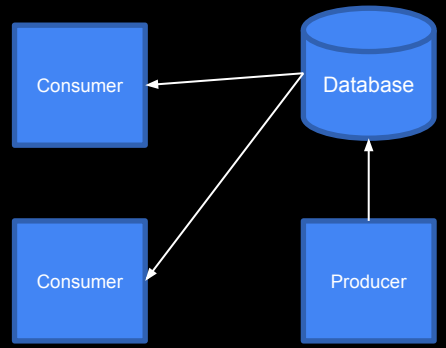
Intro: Data through HTTP

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 a consumer (front-end) will take data from the database and **consume** it by displaying it to the user.

Generally, we produce once, and consume many.
In today's world, we use HTTP methods and Rest API to make this clear.

You will learn more about this in module 5, for now, you should stick with the basic understanding 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. Although they can also be nouns, these request methods are sometimes referred to as *HTTP verbs*.

In this course, we will mainly focus on the four main methods, but there are more, which offer more complex interactions.

These are the most 4 most common methods.

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 an entity to the specified resource, often causing a change in state or side effects on the server.

PUT
The PUT method replaces all current representations of the target resource with the request payload.

DELETE
The DELETE method deletes the specified resource.

Source : HTTP request methods

Examples

Here are some examples. We normally refer to 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 on a /users.

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 thing 23 (or create **thing** 23)*
DELETE /things/23	->	Delete **thing** 23

JSON Placeholder

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:
  - userId
  - id
  - title
  - body

GET API URL for getting posts from JSON Placeholder

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

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

POST API URL for creating posts to JSON Placeholder

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

GET Data

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

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 in the console, the last 10 posts. 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.

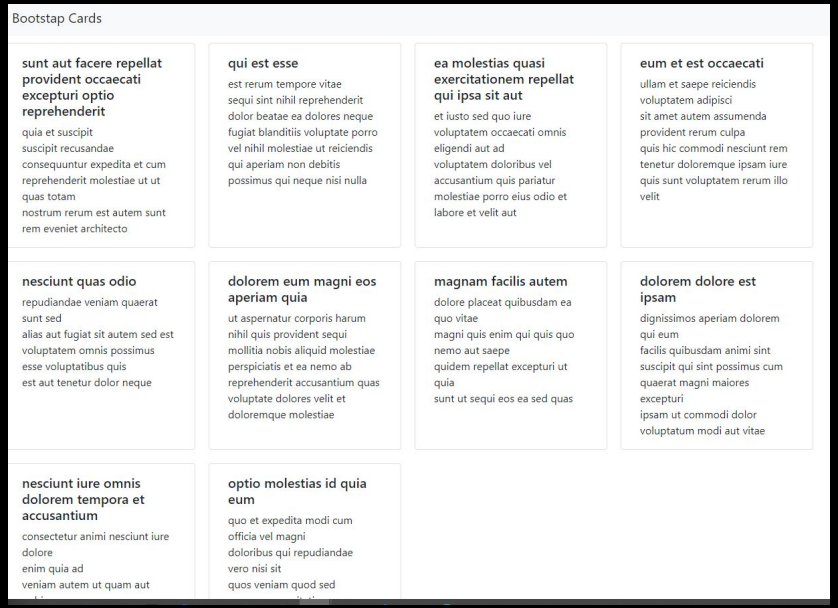
```
<html>
    <head>
        <title>JSON Placeholder Axios</title>
    </head>
    <body>
        <h1>Check the console</h1>
    </body>
    <!-- need to include the axios script first -->
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/axios/1.4.0/a
xios.min.js"></script>
    <script>
axios.get('https://jsonplaceholder.typicode.com/posts')
        .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.

Similar to 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.



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, but this property does not configure the actual representation of the animation, which 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 uses the calculated 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 a @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 running at 50% 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.

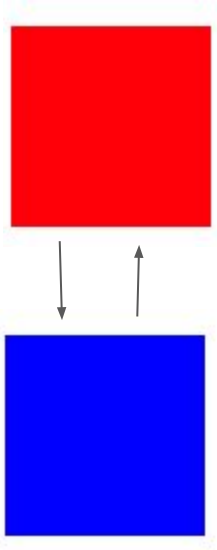
Almost all CSS properties are animatable. Color and positioning are commonly animated, as are transformations such as rotation and translation.

```
<!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>
```

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

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.



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.

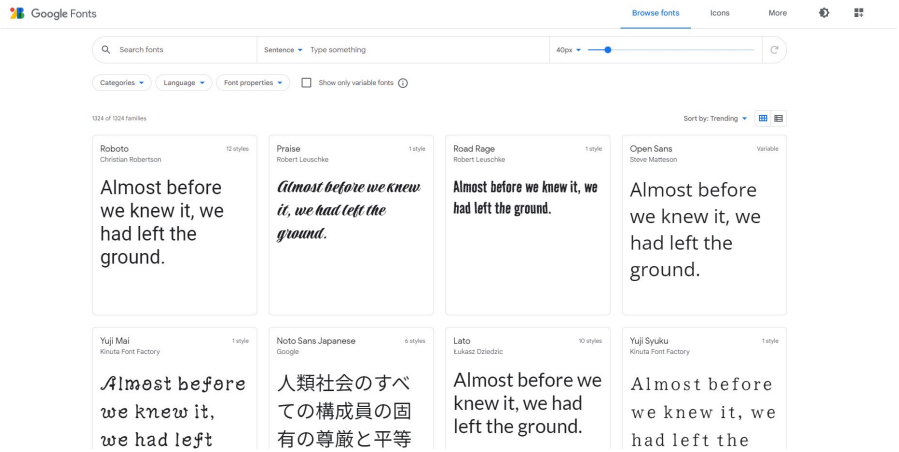
Hovering over the text in this example will gradually change the color from black to purple.

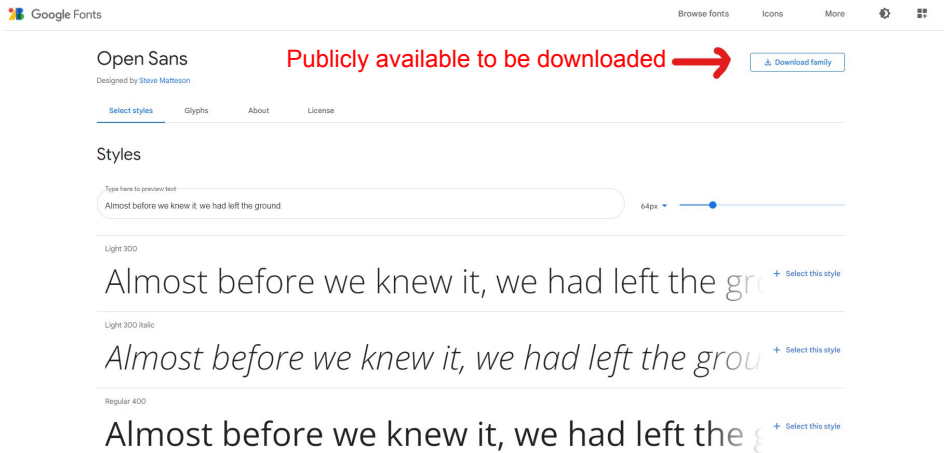
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>CSS Transitions</title>
    <style>
        p { font-family: sans-serif;
            font-size: 24px;
            font-weight: bold;
            color: black;
            transition: color 2s; }

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

See Using CSS transitions for more information

Font Libraries

- **Google Fonts:** Google Fonts is a font embedding service library which contains a vast library of publicly available fonts.

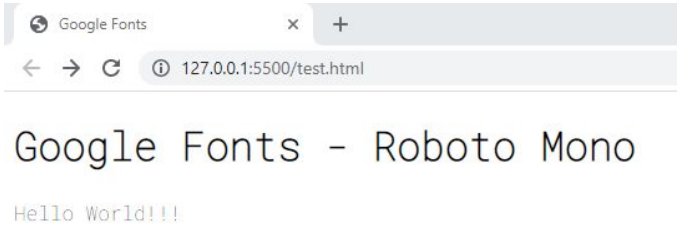




Google Fonts example

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

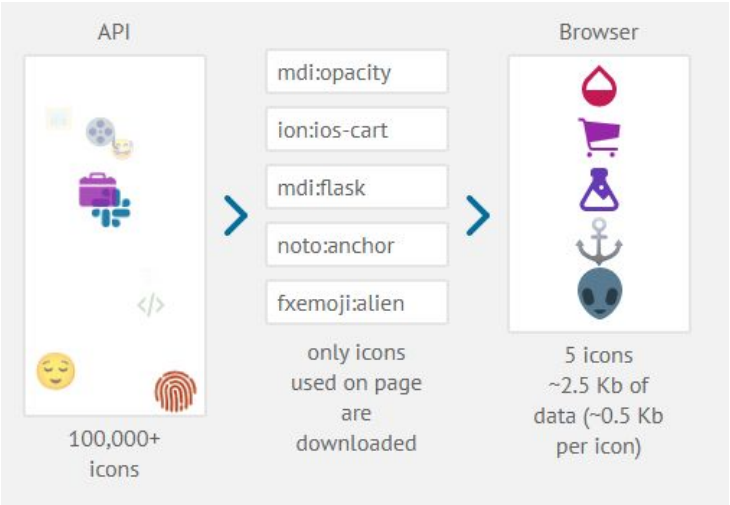
- Reference the font you want to use from Google Fonts.
- Add CSS rules to specify font-families.




```
<!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>
```

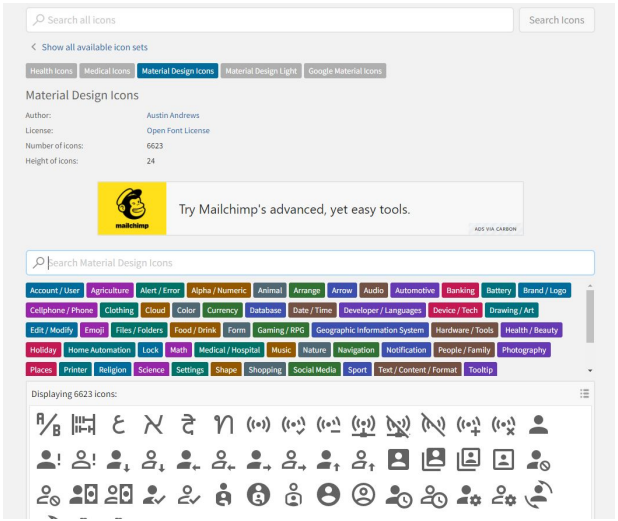
Icon Libraries

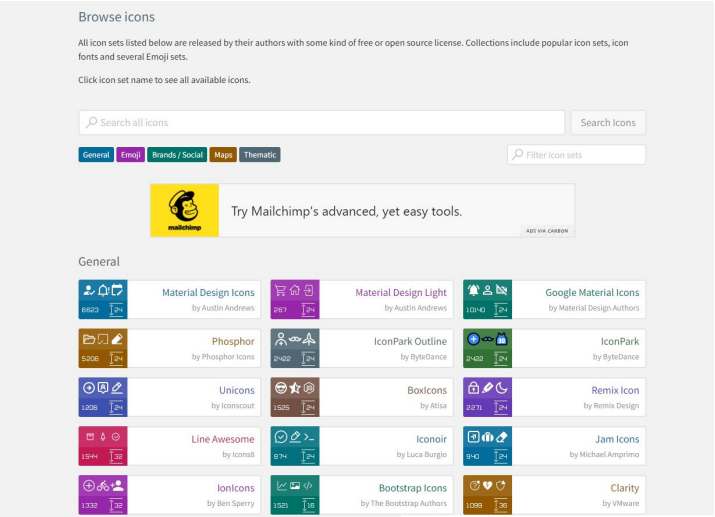
- **Iconify.design**: Iconify.design is a unified icons framework which consists of over 100,000 vector icons.
  - Various collections to choose from.
  - Easy to use.

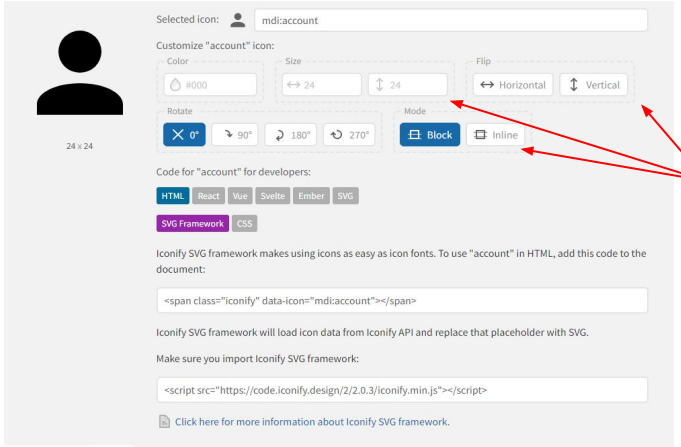


Icon Libraries

Numbers of icons with different designs can be used by simply copy and pasting from the **Iconify** website.





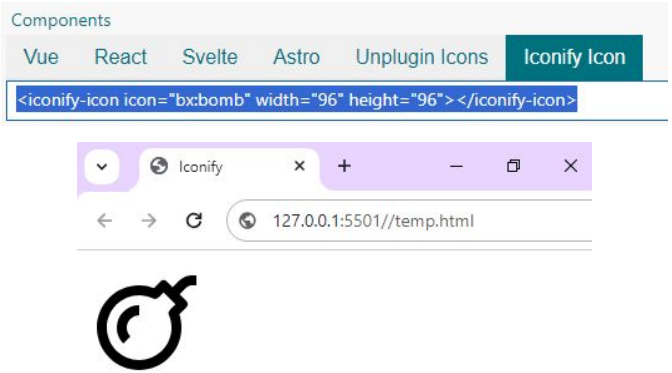


Customisable on the website before copy and paste in your code

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

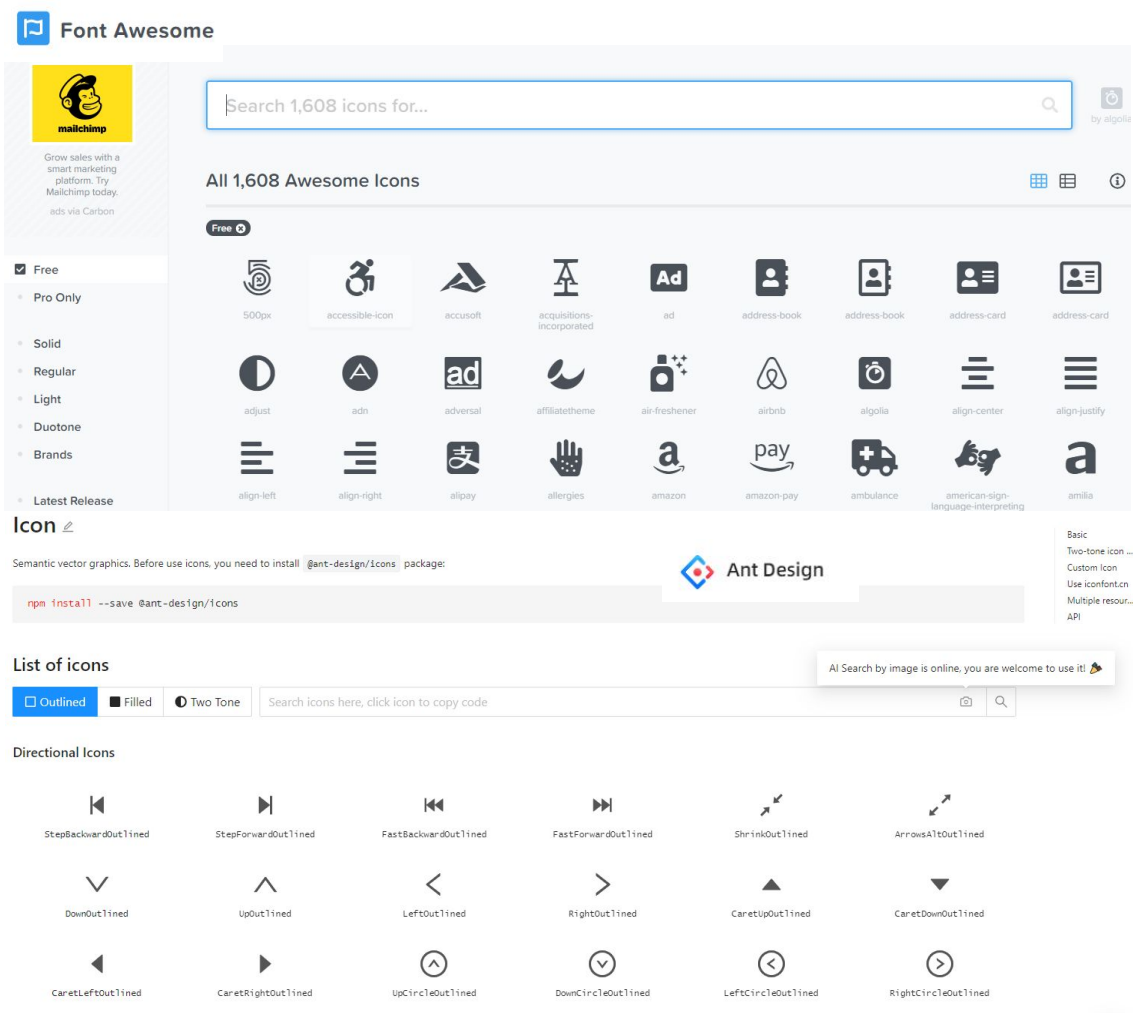



```
<html>
  <head>
    <meta charset="utf-8" />
    <title>Iconify</title>
    <!-- Import Iconify Framework from
https://iconify.design/docs/iconify-icon/#register
ing-the-web-component -->
    <script
src="https://code.iconify.design/iconify-icon/2.0.
0/iconify-icon.min.js"></script>
  </head>
  <body>
    <!-- Add icon using Iconify Icon component -->
    <iconify-icon icon="bx:bomb" width="96"
height="96"></iconify-icon>
  </body>
</html>
```

Icon Libraries

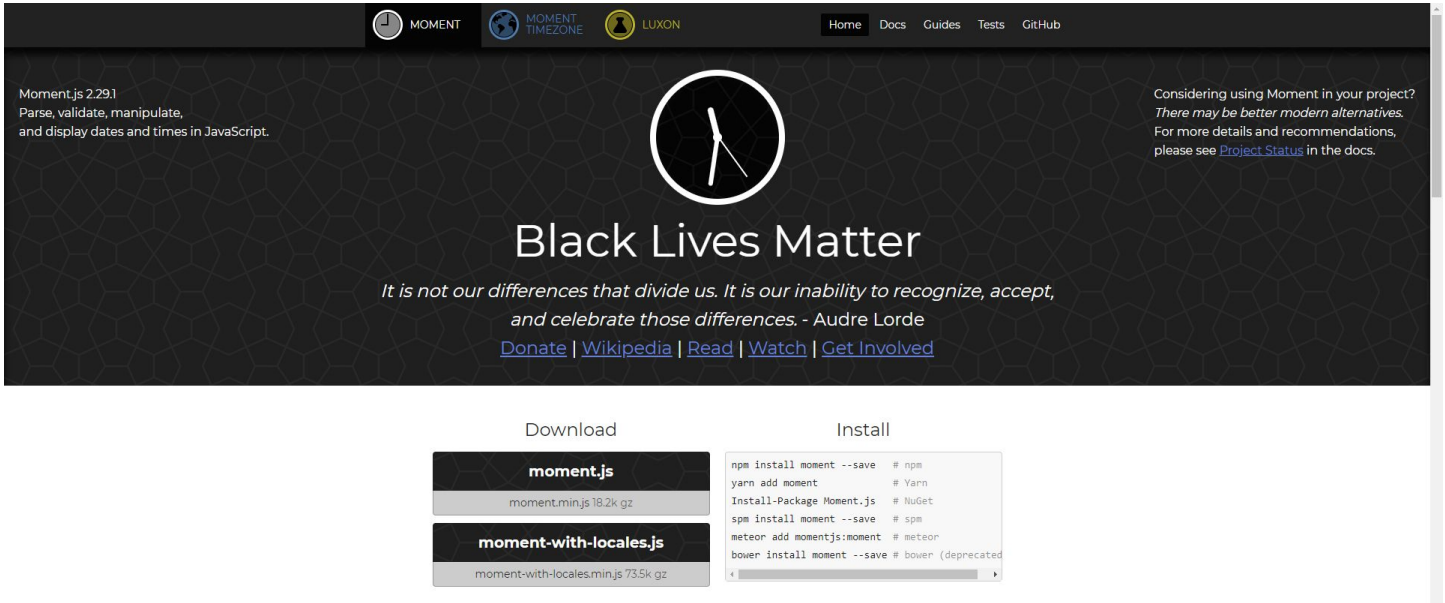
Other good icon libraries:

- Ant.design
- Font Awesome



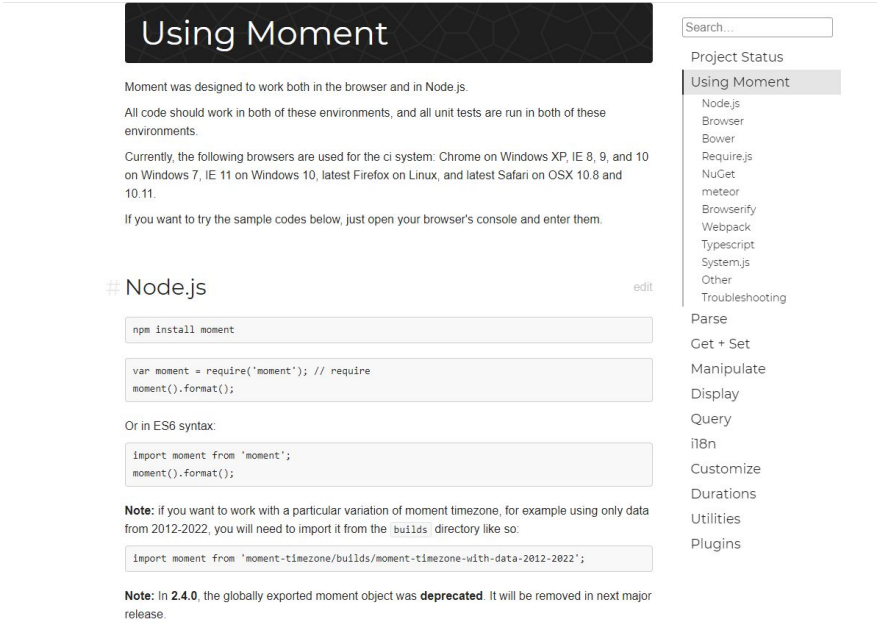
DateTime and Internationalisation libraries

- **MomentJS:** One of the most used JavaScript date libraries among software engineers. However, over the past few years, other alternatives challenged its existence.



DateTime and Internationalisation libraries

- Supports a wide range of languages and libraries.
- **MomentJS** is a legacy project now focusing on **stability**. It is **not dead** but is indeed **done**.



Using Moment

Moment was designed to work both in the browser and in Node.js.

All code should work in both of these environments, and all unit tests are run in both of these environments.

Currently, the following browsers are used for the ci system: Chrome on Windows XP, IE 8, 9, and 10 on Windows 7, IE 11 on Windows 10, latest Firefox on Linux, and latest Safari on OSX 10.8 and 10.11.

If you want to try the sample codes below, just open your browser's console and enter them.

Node.js edit

```
npm install moment
```

```
var moment = require('moment'); // require
moment().format();
```

Or in ES6 syntax:

```
import moment from 'moment';
moment().format();
```

Note: if you want to work with a particular variation of moment timezone, for example using only data from 2012-2022, you will need to import it from the `builds` directory like so:

```
import moment from 'moment-timezone/builds/moment-timezone-with-data-2012-2022';
```

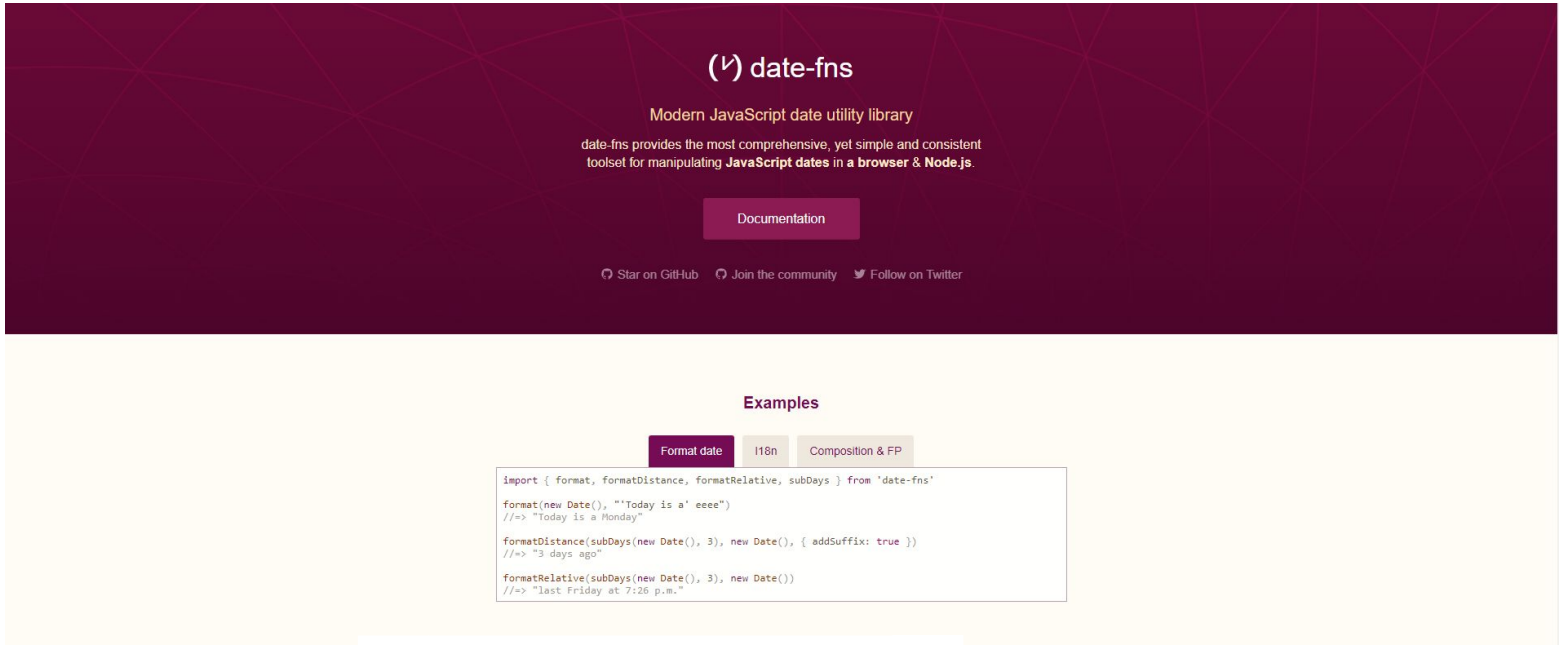
Note: In **2.4.0**, the globally exported moment object was **deprecated**. It will be removed in next major release.

Search...

Project Status
Using Moment
Node.js
Browser
Bower
Require.js
NuGet
meteor
Browserify
Webpack
Typescript
System.js
Other
Troubleshooting
Parse
Get + Set
Manipulate
Display
Query
i18n
Customize
Durations
Utilities
Plugins

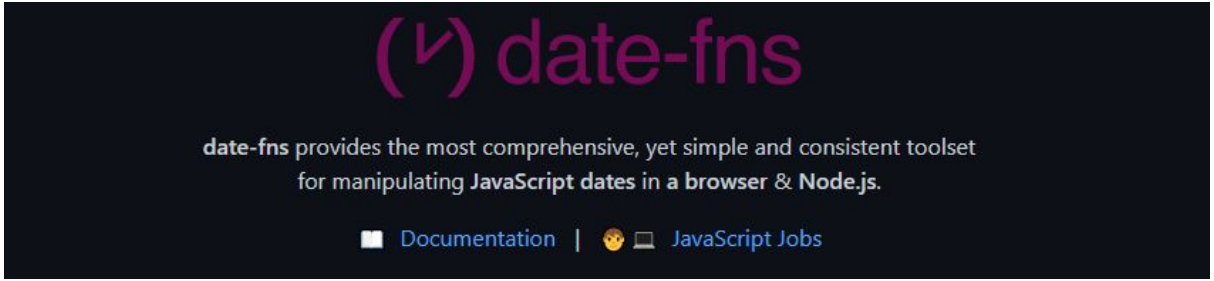
DateTime and Internationalisation libraries

- **Date-FNS:** Date-FNS is similar to Moment JS but is written with a more modular approach.



DateTime and Internationalisation libraries

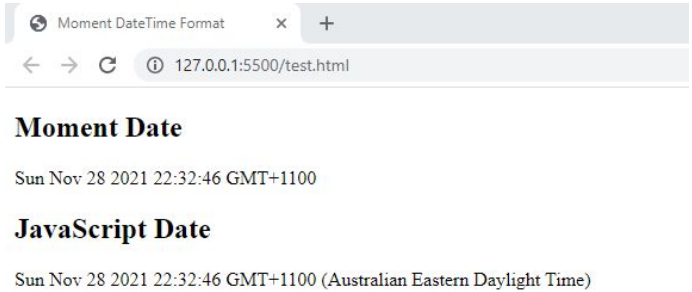
- A modern JavaScript date utility library.
- Well maintained and developed.
- Multiple benefits:
  - Modular
  - Fast.
  - Consistent.
  - I18n.
  - Immutable & pure
  - etc.



Moment.js example

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

- Import Moment library.
- Create the function to format the date/time.
- Place the result into the place you want to render the result.




```
<html>
<head>
  <title>Moment DateTime Format</title>
  <!-- reference Moment.js library -->
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.29.4/moment.mi
n.js"></script>
</head>
<body>
<h2>Moment Date</h2>
<!-- container for Moment.js output -->
<div id="displayMoment"></div>

<h2>JavaScript Date</h2>
<!-- container for JavaScript Date output -->
<div id="displayJsDate"></div>

<script type="text/javascript">
(function() {
  // instantiate a moment object
  let nowMoment = moment();

  // instantiate a JavaScript Date object
  let nowDate = new Date();

  // display value of moment object in #displayMoment div
  let eDisplayMoment = document.getElementById('displayMoment');
  eDisplayMoment.innerHTML = nowMoment;

  // display value of Date object in #displayJsDate div
  let eDisplayDate = document.getElementById('displayJsDate');
  eDisplayDate.innerHTML = nowDate;
})();
</script>
</body>
</html>
```

Lab #8: Moment.js

Use the documentation at Moment.js to help in answering these exercises.

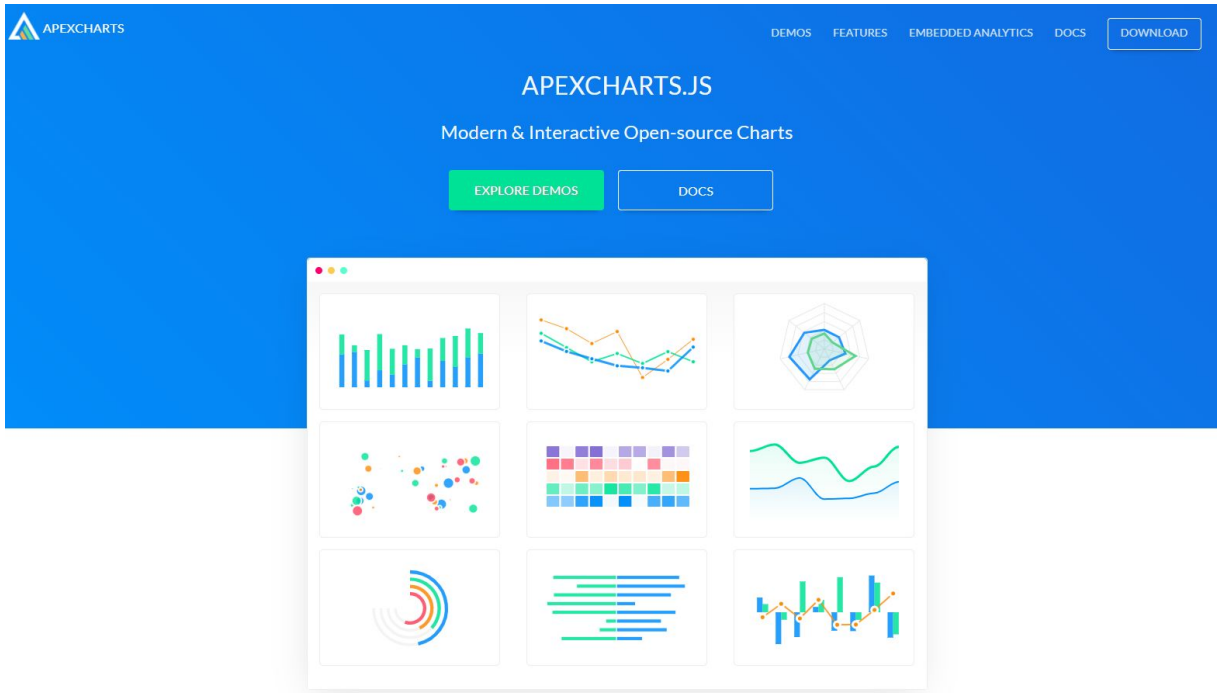
Using the Moment.js library, try to solve the below problems:

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

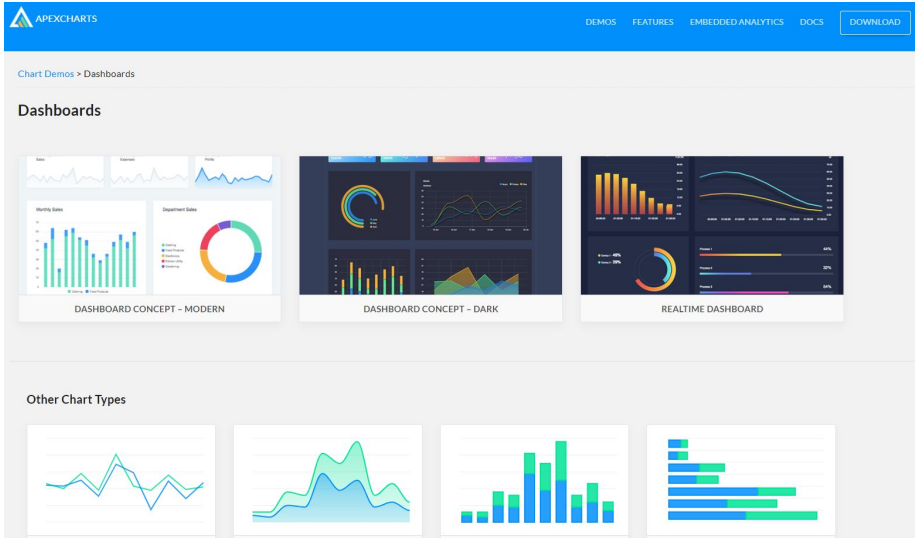
Charts libraries

- **Apex Charts:** A modern and interactive open-source charts library.



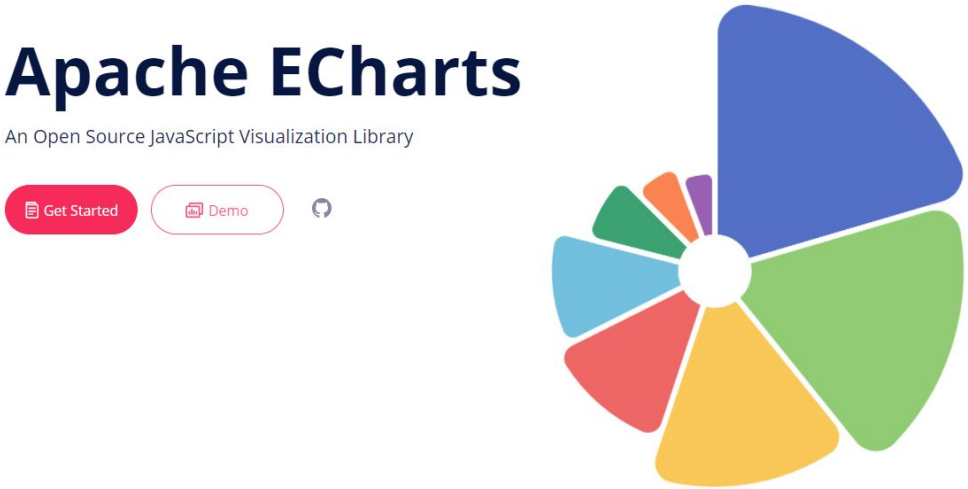
Charts libraries

- **Apex Charts** look great on different devices, and the library allows for customisation and comes with comprehensive documentation.
- Features:
  - Responsive.
  - Interactive.
  - Dynamic
  - High Performance.
- However, it can be laggy with larger datasets



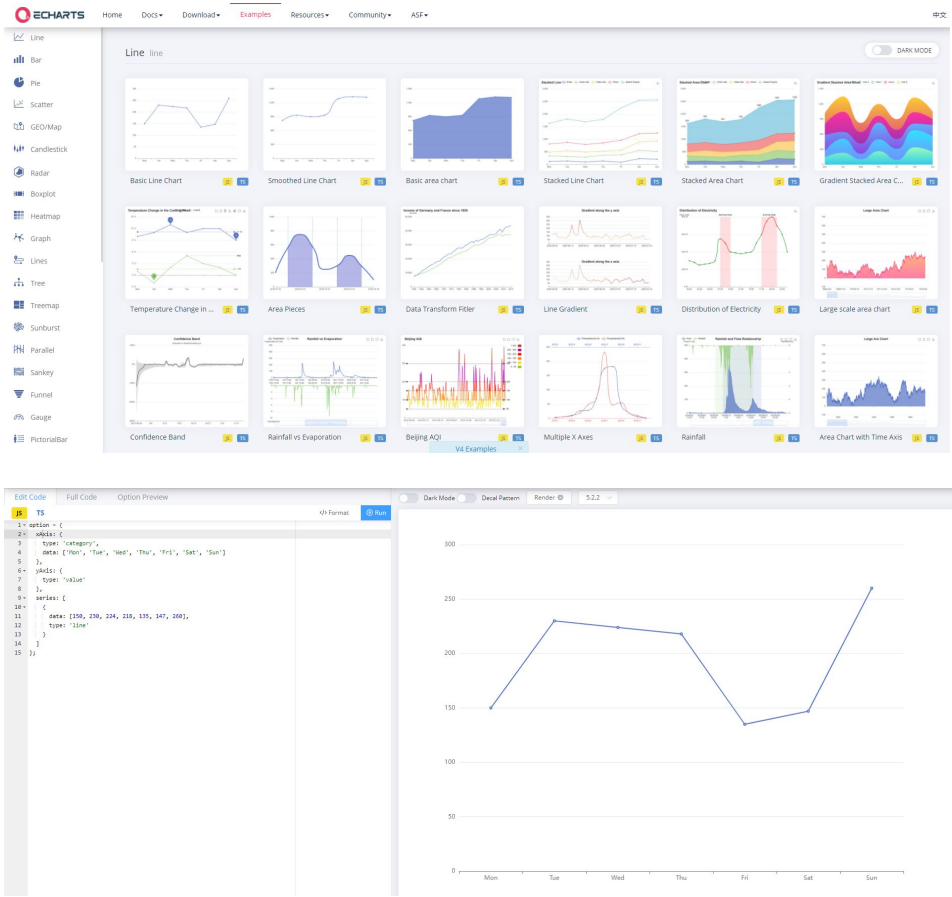
Charts libraries

- **Apache Echarts:** is a huge open-source JavaScript library for data visualisation developed by Apache.



Charts libraries

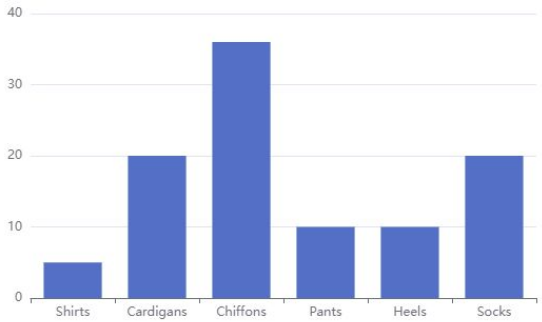
- **Apache Echarts** is super useful for JavaScript data visualisations intended for the Web.
- Works great with big datasets.
- It also supports both SVG and Canvas rendering.
- It also provides comprehensive interactable documents with visual examples.



eChart Example

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

- Select a Div where to place the chart
- Create a chart object
- Pass the data to the chart object and render it using the setOption method




```
<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">
      // Initialise 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);
            })

        // Initialise 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 drop down selection to choose individual product categories
- (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