

Welcome!

# Agenda

- Review
- Terminal
- Asynchronous Programming
- Synchronous Programming
- (Promises)
- (APIs)
- (AJAX)

Review

# Terminal

# What is the terminal?

- A way to manipulate and interact with your computer
- It's entirely text-based
  - As opposed to the regular way we interact with our computers - called WIMP (Windows, Icons, Menus and Pointers)

# Why use the terminal?

- It's very fast
- It's automatable and flexible
- No interruptions
- Sometimes it is the only way
  - Command Line Interaction (CLI)
  - Web Servers
- It gives us access to tools we wouldn't have access to otherwise

# How do we use the terminal?

- We will open up a Terminal application
  - Windows: Git Bash, Cygwin, WSL etc.
  - Mac: **iTerm2**, Terminal etc.
- Once there, we interact with a **shell** (we will be using the Bash shell)
  - This, more or less, sends commands directly to the **kernel** (think of this as the hardware)

# The Bash Shell?

- Bash is a regular program on your computer
- It was created to take commands from you
  - We talk to it using the **Bash Shell Language**



# Are there other shells?

Yes!

- Bash (Bourne Again Shell)
- C Shell
- Z Shell (zsh)
- Korn Shell
- Bourne Shell
- Debian's Almquist Shell (dash)
- Plus many, many more

# What can you do with it?

Anything!

- Run programs to make all sorts of changes
- Editing files and images
- Converting files between types
- Creating back-ups
- Interacting with databases
- Making and copying files and folders
- Downloading, compiling and running programs
- Plus anything else you can think of!

# How do you work with it?

- **Interactively**
  - Opening up a REPL (like Git Bash or iTerm2)
- **Non-interactively**
  - Running scripts

# Some Utilities

- Who am I?
- Where am I?
- Show me everything that is running?
- Show me the manual?
- Opening files

# Common Commands



```
pwd          # Where am I?

ls           # List all files in the current directory (folder)

cd           # Change directory (folder)

mkdir        # Make a directory (folder)

rm           # Remove a file or a directory (folder)

cp           # Copy a file or a directory (folder)
```

# Flags and Arguments



```
cd Desktop      # Move into the desktop directory
```

Desktop, in this case, is an argument



```
rm -r FolderToBeDeleted
```

```
# Delete a folder and recursively delete all files and folders within it
```

**-r**, in this case, is a flag. It gives the command a few more details on how to run

# Common Commands



```
touch      # Create a file
open       # Open a file in the default application
clear      # Clear your screen
cat        # Print the contents of a file
man        # Show the manual for a given command
```

# Resources

- Watch these Code Academy videos
- Read these
  - Quick Left's Tutorials - start from the bottom!
  - Learn CLI the Hard Way
- Track down the Terminal City Murderer
- Some other useful links
  - 40 Terminal Tricks and Tips
  - 25 Useful Find Examples
  - Terminal Cheatsheet



Asynchronicity

# Asynchronous vs Synchronous

- Programming languages are either synchronous or asynchronous
- Synchronous languages are where tasks are performed one at a time. You need to wait for a task to finish before moving on
- Asynchronous languages allow you to move on to another task before the previous one finishes
- This is something we have to deal with in JavaScript (as it is asynchronous)
  - Because some things take time!

# What takes time?

- API Calls
- Events
- User Interactions
- Animations

# How do we deal with asynchronicity?

- Callbacks
  - We have seen them
- Promise
  - We are about to see them
- `async` and `await`\*
  - A little too new for us to use straight away. We will see them later

# Promises

# What are promises?

- Promises represent eventual results of an **asynchronous** operation
- "A Promise is an object representing the eventual completion or failure of an **asynchronous** operation"
- It's an object that may produce a single piece of data at some point in the future
  - Either a **resolved** value
  - Or a **rejection** (an error that tells us why it wasn't resolved)

# States and Fates

- Promises have three mutually exclusive potential *states*:
  - **Fulfilled**: The action relating to the promise succeeded
  - **Rejected**: The action relating to the promise failed
  - **Pending**: Hasn't fulfilled or rejected yet
- We say that a promise is "**settled**" if it isn't pending

# A little like event listeners, but...

- A promise can only succeed or fail once
- If a promise has succeeded or failed and you later add a success/failure callback, the correct callback will be called (even though the event happened earlier)



# Why use promises?

- They help us write readable code
- They help us deal with the complexities of asynchronous programming
- They help us avoid "callback hell" or "the pyramid of doom"
- They are the backbone of some of the newer features coming out with JavaScript (e.g. `async` and `await`)
- Lots of libraries/frameworks/packages use promises (so we will have to be the consumers of them anyway)

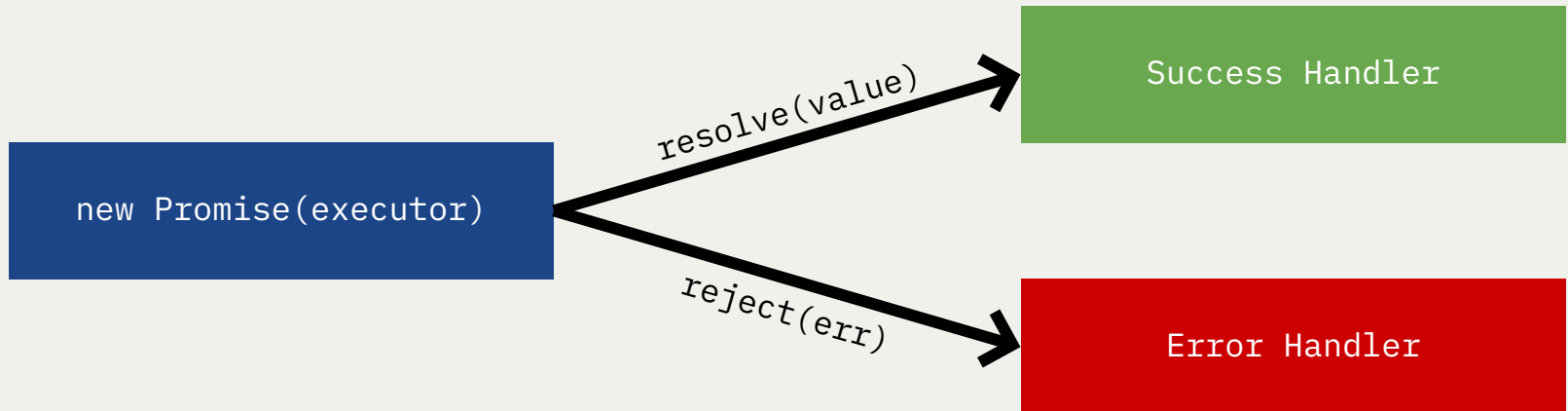
# States and Fates

- Promises have two mutually exclusive potential *fates*:
  - **Resolved**: Finished (or locked into a thenable or another promise)
  - **Unresolved**: If trying to resolve or reject will make an impact

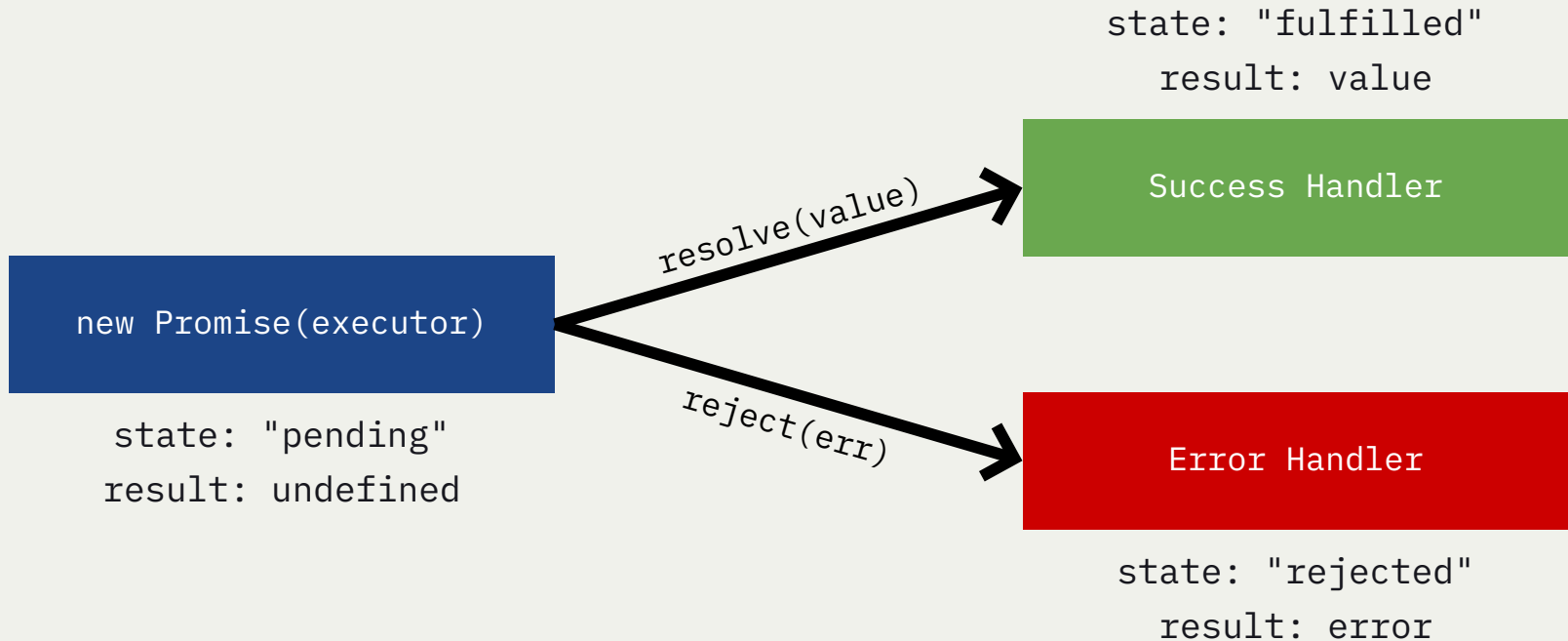
# Some Terminology

- **Executor**: A function that contains the producing code
- **Fulfilled**: Succeeded
- **Rejected**: Failed
- **Pending**: Waiting
- **Settled**: Not pending
- **Resolved**: Finished
- **Unresolved**: If trying to resolve or reject will make an impact
- **Thenable**: A piece of data that is promise-like (it has a `.then` method)

# How do they work?



# How do they work?




# Creating Promises

The executor callback function automatically receives a **resolve** function and a **reject** function

```
function myExecutor(resolve, reject) {  
  if (true) {  
    resolve("This will run the .then method");  
  } else {  
    reject("This will run the .catch method");  
  }  
}  
  
let myPromise = new Promise(myExecutor);
```

# Consuming Promises


The data you provide to the **resolve** function will be passed to the **.then** callback function



```
function myExecutor(resolve, reject) {  
  if (true) {  
    resolve("This will run the .then method");  
  } else {  
    reject("This will run the .catch method");  
  }  
}  
  
let myPromise = new Promise(myExecutor);  
  
function successHandler(data) {  
  console.log(data);  
}  
  
myPromise.then(successHandler);
```

# Consuming Promises

The data you provide to the **reject** function will be passed to the **.catch** callback function



```
function myExecutor(resolve, reject) {
  if (true) {
    resolve("This will run the .then method");
  } else {
    reject("This will run the .catch method");
  }
}

let myPromise = new Promise(myExecutor);

function successHandler(data) {}
function errorHandler(error) {}

myPromise.then(successHandler).catch(errorHandler);
```



# Exercise

Turn a timer into a promise.

I would like to be able to write the following lines of code:



```
function afterASecond() {  
  console.log("That has worked");  
}
```

```
delay(1000).then(afterASecond);
```

```
// You'll have to use .setTimeout somewhere!
```

```
// Also, this is very hard. Please don't be discouraged by it!
```

# Exercise

Turn an event into a promise.

I would like to be able to write the following lines of code:



```
function onHeadingClick() {  
  console.log("That has worked");  
}
```

```
onClick("h1").then(onHeadingClick);
```

```
// You'll have to use .addEventListener somewhere!  
// Try to add an event handler that will only run once (use MDN)!  
// Also, this is very hard. Please don't be discouraged by it!
```

# My (biased) advice?

- You'll rarely have to make your own promises from scratch, but you will regularly consume promises
  - What I mean by this is to focus on the concept of Promises, and try to get used to writing and understanding `.then` and `.catch` handlers
  - We will get lots of practice with that in the following classes!

# Resources

- [MDN: Promises](#) and [MDN: Using Promises](#)
- [Google Web Fundamentals: Promises](#)
- [JavaScript.info: Promises](#)
- [Scotch: JavaScript Promises](#)
- [David Walsh: Promises](#)
- [You Don't Know JS: Promises](#)
- [Exploring JS: Promises](#)
- [Eric Elliot: Promises](#)
- [Domenic: The Point of Promises](#)

APIs

# What are APIs?

- It stands for Application Programming Interface (API)
- Software that allows two programs to communicate with each other
  - It all starts with shared data
- The principle of API abstraction allows for decoupling applications
- Can be private or public, internal or external, and paid or free

# What are APIs?

- An API is a set of rules that allows one piece of software to talk to another

# What is an API?

- **Application** - Any application
- **Programming** - The engineering part that translates given inputs into outputs
- **Interface** - The interface, the way we interact

Picture an ATM!



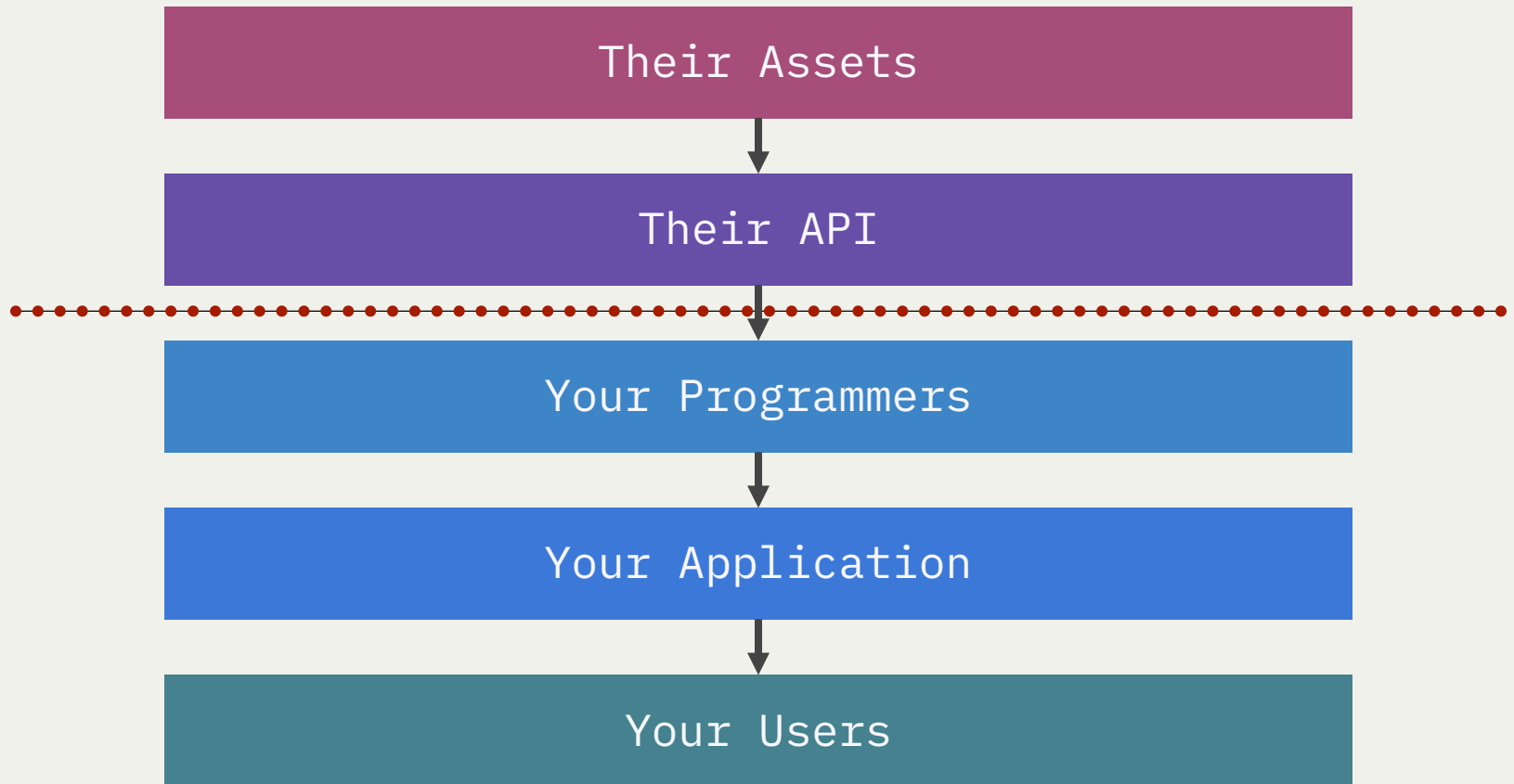
# Types of APIs

- Operating Systems
- Remote APIs
- Libraries, Frameworks and Software Development Kits (we will focus on this)
- Web APIs (we will focus on this)

# For us though?

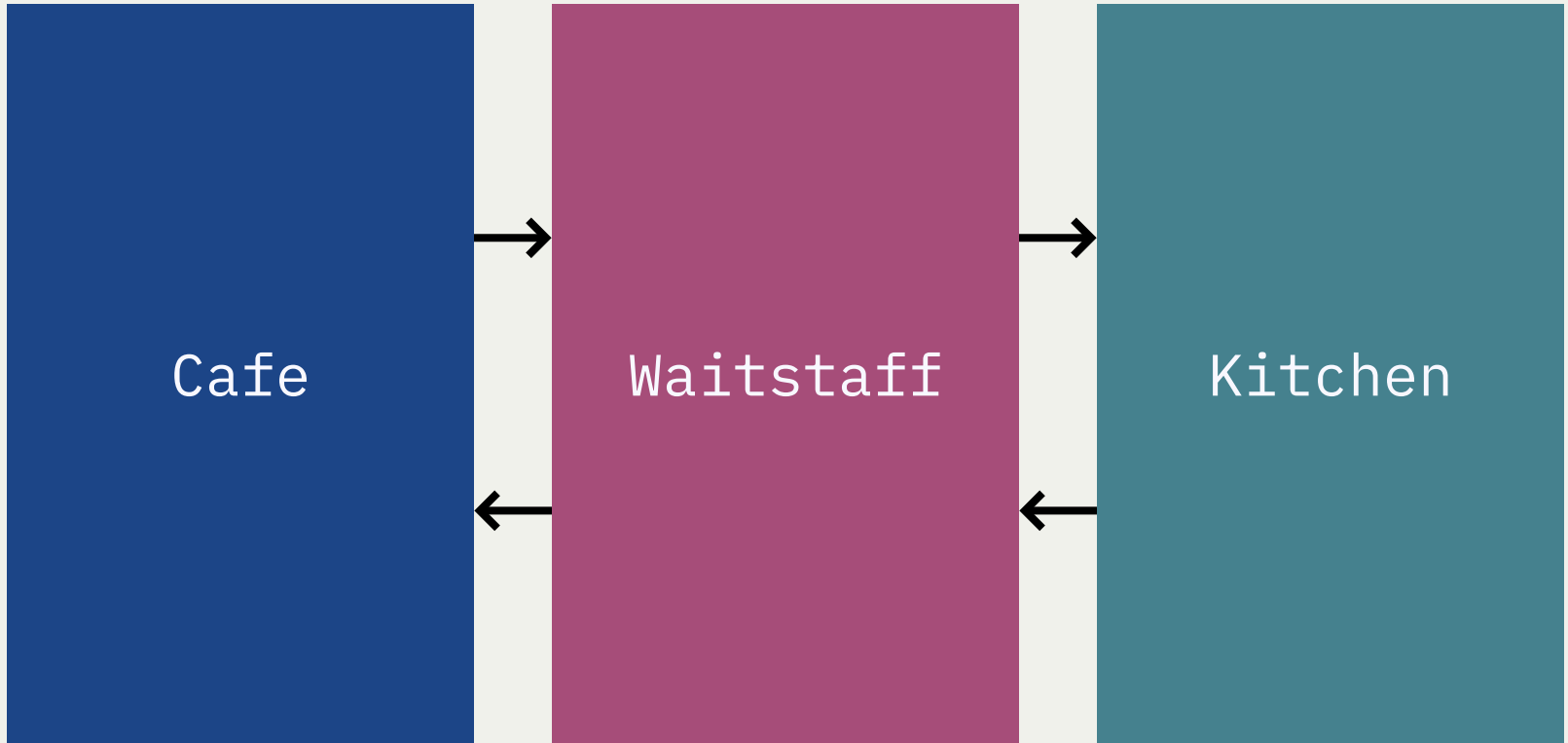
- There are browser APIs (think things like the DOM API)
- There are third-party APIs - where we access data from a remote server

# How do they work?



# How do they work?

- **Assets** - Anything that is chosen to be shared (data, processes etc.)
- **API** - The gateway to those assets
- **Developers** - The API is exposed to your developers
- **Your Application** - Your developers code your application, and it is powered by the API
- **Your Users** - Your users use the app that is created and reap the benefits

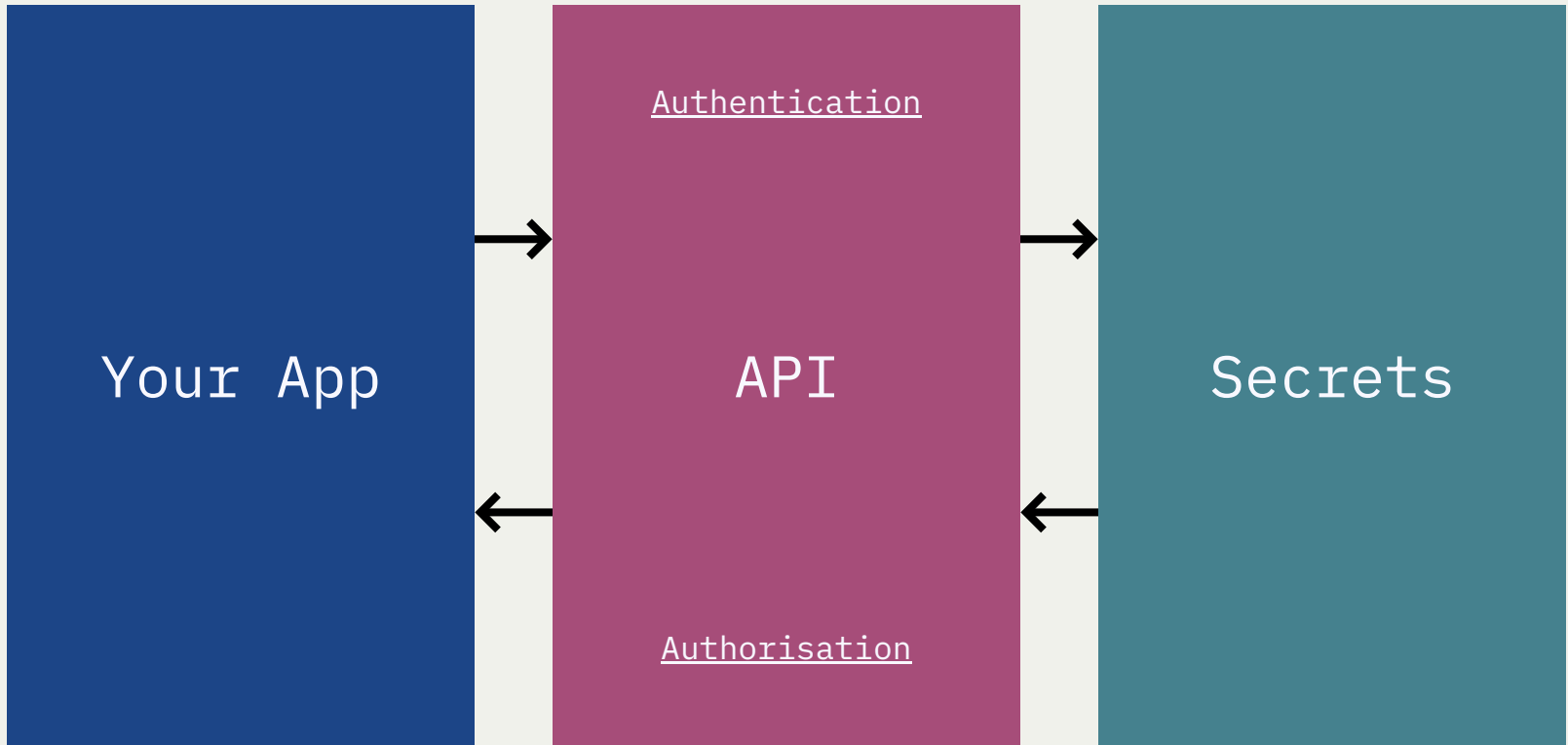


```
graph LR; App[Your App] --> API[API]; API --> Secrets[Secrets]; Secrets --> API; API --> App;
```

Your App

API

Secrets



# Benefits of APIs for Providers

- APIs let you build one app off another
- APIs are like a universal plug
- APIs can be profitable
- APIs can allow you to decouple code - can make your applications more performant and finding developers easier
- APIs can essentially outsource complexity
- API providers can promote your application



# Benefits of APIs for Consumers

- APIs can allow you to get data, and add features, that would otherwise be difficult and time-consuming
- APIs can make your app realtime
- APIs can introduce similar flows to an application (e.g. logins etc.)
- Apps using APIs can provide useful starting data for users

# Downsides

- Building them
- Maintenance
- Hosting
- Documentation
- For developers - future support
- Reliance on other services

# Authentication and Authorisation

## Authentication

Are you allowed to access the system?

## Authorisation

What can you access in the system?

Often we need to sign up to APIs

# Internal Workings

# What do we need to talk to APIs?

- **The endpoint URL**
  - Where are we getting the data from?
- **HTTP Method**
  - Describes our intentions (e.g. are we asking to GET data, are we asking to POST data etc.)
- **HTTP Headers**
  - Key-value pairs that provide data to the client and the server (e.g. for authentication)
- **Body Data**
  - Any data that you wish to send

# URLs (often called endpoints)



```
https://subdomain.domain.com:80/path/to/resource?key1=value1&key2=value2
```

- https:// - Scheme
- subdomain - Subdomain (occasionally there)
- domain.com - Domain or Host
- :80 - Port (not often there)
- /path/to/resource - Path
- ?key1=value1&key2=value2 - Query String or Parameters

# HTTP Methods

- There are lots of different types of methods, but the main ones are:
  - **POST** - Creating data
  - **GET** - Reading data
  - **PUT / PATCH** - Updating data
  - **DELETE** - Deleting data
- This is called **CRUD** (Create, Read, Update, Delete)

# HTTP Headers

- Key-value pairs
- Used to share information between the client and the server
- You can store things like cookies, metadata and authentication tokens in here
  - The most common though is to describe what data you'd like back using "Content-Type"
- Here is a list of all valid headers



# Body Data

- The body contains any data that you'd like to provide to the server
  - Usually as a JSON-encoded string (more on that soon)

# Responses

Once the request has been made the server will send back an HTTP response, which is made up of data and a status code. There are lots of different status codes, but they are somewhat organised:

- **100+:** Informational Response
- **200+:** Request has succeeded
- **300+:** Request has been redirected
- **400+:** An error on the client's end has occurred
- **500+:** An error on the server's end has occurred

AJAX

# What is AJAX?

- **A**synchronous **J**avaScript & **eX**tensible **M**arkup Language
- It is a way to make your pages live
  - You can talk to other servers while you are still on the page
- It is a technique to send and retrieve information behind the scenes without needing to refresh the page

# Where is AJAX used?

- In feeds (such as Twitter)
- Chat rooms and messaging apps
- For voting and rating
- Autocompletion
- Form submission and validation
- To access data that you don't have
- To show extra information
- In games (e.g. to save scores)

# Why is AJAX so useful?

- It makes your pages live
- It is much faster
- It tends to give a greater user experience
- It is fancy
- It is popular in the workplace
- It is the foundation of things such as APIs

# How do we work with it?

- XMLHttpRequest
- Fetch

# One Thing!

- To make API requests, we need a server (AJAX doesn't work on a file URL)
- Install HTTP-Server (you'll need Node first!)
- Restart the terminal
- To start the server: run the following command in the directory you want to serve and open the URL it provides
  - `http-server .`



# What is fetch?

- It is a way to make AJAX Requests
- It is a function that is defined automatically for us
- It is supported by all major browsers now
  - For those that don't support it, there is a polyfill

# What is fetch?

- You make an *HTTP Request* with it
  - Specifying the URL, the method etc.
- It comes back with an *HTTP Response*
  - Most of the time, the data is returned as **JSON**
- That returns a *Promise*
  - We can work with the returned data in a *.then*
  - We can handle errors in a *.catch*

# What is JSON?


- It stands for JavaScript Object Notation
- A format for storing and sharing information
- It looks almost exactly like JavaScript Objects
  - *But keys are quoted*
- It's a format that is really easy to work with, particularly in JavaScript:
  - We can convert an object into JSON using:  
`JSON.stringify(obj);`
  - We can convert JSON into an object using  
`JSON.parse(json);`

# Using Fetch



```
fetch( URL, HTTP_OPTIONS? )  
  .then( SUCCESS_HANDLER )  
  .catch( ERROR_HANDLER );
```

# Using Fetch



```
fetch("http://api.open-notify.org/astros.json")
  .then(function(res) {
    return res.json();
  })
  .then(function(data) {
    console.log(data);
  });
```

- We have make a Request to the API
- We parse the Response and turn it into a JS Object with `.json()` (this uses `JSON.parse` internally)
- We can then work with the data!

# Parameters

- We can attach a Query String or Parameters to a URL
- This provides extra information to the API
- Works in a similar way to an object - query strings have keys and values
- Looks something like this:



```
?keyOne=valueOne&keyTwo=valueTwo&keyThree=valueThree
```

# Using Fetch



```
fetch("https://randomuser.me/api/?results=10")  
  .then(function(res) {  
    return res.json();  
  })  
  .then(function(data) {  
    console.log(data);  
  });
```

# Using Fetch



```
fetch("https://randomuser.me/api/?results=10&gender=male")
  .then(function(res) {
    return res.json();
  })
  .then(function(data) {
    console.log(data);
  });
```



# OMDB API

- Go to OMDB API's Website
- Get an API Key from here and select Free
- Input your details
- Check your email
- Click the verify API key link
- Copy your API key from that email

# OMDB API - Authentication



```
fetch("http://www.omdbapi.com/?t=Jaws&apikey=API_KEY&plot=full")
  .then(function (res) {
    return res.json()
  })
  .then(function (data) {
    console.log(data)
  });
```

# OpenWeatherMap

- Go to the [OpenWeatherMap API website](#)
- Sign up for an [API key here](#)
- Fill in your details
- Log in
- Go to the [API Key Tab](#) on your settings page
- Copy the API Key
- It'll take ten minutes for the API Key to work

# OpenWeatherMap



```
let baseURL = "http://api.openweathermap.org/data/2.5/weather";  
let parameters = "?q=Sydney&units=metric&appid=API_KEY";  
  
fetch(baseURL + parameters)  
  .then(function (response) {  
    return response.json();  
  })  
  .then(function (data) {  
    console.log(data);  
  });
```

# Giphy API

- Open up the documentation here
- Create an account here
- Create an app here
  - Copy your API key!
- Looks at the Docs here

# Yandex Translate API

- Open up the documentation [here](#)

# Some other things

- Using Geolocation
  - getCurrentPosition
- Speech to Text, Text to Speech
  - SpeechRecognition
  - SpeechSynthesis
- Air Quality Index
- Rest Countries

# Resources

- [Rapid API](#)
- [MDN: Using Fetch](#)
- [CSS Tricks: Using Fetch](#)
- [Scotch.io: Fetch](#)
- [David Walsh: Fetch](#)
- [Google Developers: Fetch](#)
- [Google Developers: Working with the Fetch API](#)
- [MDN: Fetch API](#)



# Homework

- Do more with APIs!
- Track the International Space Station's Position
  - Using [this API](#)
  - Show the current location of it in your HTML
  - Bonus: Link it up to the Google Maps API!
- Or anything other API you want
- ***Note: Don't use ones requiring OAuth***
  - We will be going through this later

# What's next?

- More APIs
- More AJAX

Thank you!