Summary:

The Internet

Breaking down how the Internet works (in broad strokes).

HTTP:

Hyper-Text Transfer Protocol (HTTP):

1. You send a request for information (i.e. by going to a website, or from script/link tags, or using fetch)
2. That request is sent to a DNS lookup, that has a list of IP addresses and matching URLs.
3. It will forward your request to the correct IP address (a server - which is just a way of saying a computer somewhere else).
4. That server will process your request and send a response with the data via the DNS lookup
5. Which will send it back to your IP address (your computer).

What kind of information can we transfer?

- HTML - Hyper-text markup languages ("Hyper" == on the internet)
- Fonts
- Images
- JavaScript
- libraries and tools...
- Data

Your websites are already doing this. Large websites can end up doing tens, even hundreds of requests just to load the page.

End-to-end encryption

People can sit on your network, watching the activity, watching data you send, and even changing the data you receive (man. Your information is encrypted so long as you're using the HTTPS type of URL. That means that anyone watching your traffic cannot see what you are sending and receiving.

SSL Certificate

When you build a website you register for an SSL certificate, it's a way of saying who a website belongs to. People can use this as evidence that you are the website you claim to be.

- Certificates can be free.
- The sign-up process is really, really easy (there are lots of easy websites for it)

Because of those two reasons, it's still not foolproof, and it's important to verify your URL is really important scenarios (e.g. banking).

Tangent: URLs can contain characters that look identical to Latin alphabet characters but that are actually not, and they can be registered to a different person than you think.

How we transfer data

The two main ways of sending information/data on the internet, are using these two languages:

- JSON
- (more rarely) XML

We need to use a greed-upon languages to communicate between different computers, so that we know how to structure and read complicated data.

API(Application Processing Interface):

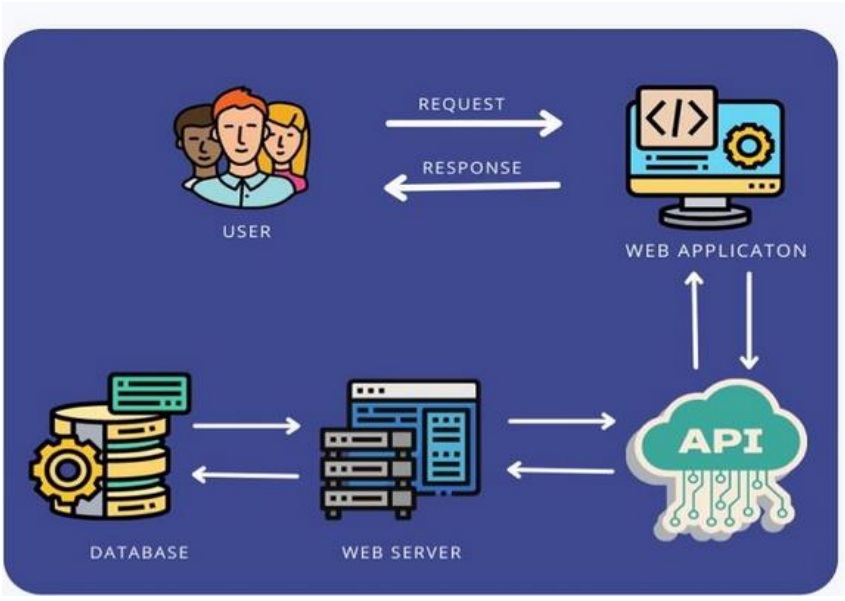
Simplest Example:

Suppose your program is a robot and you want your robot to get into a car and drive across the city. When you drive a car, do you care how it works inside? Do you care if it's a V6 or V8?diesel, or hybrid? All of those things affect how the car works, but those aren't the *interface.*

The interface of a car is the steering wheel, gearshift, and Pedals plus all of the other levers and controls to access all of the car's other functions. So if you were hypothetically building a robot to control a car, you don't have to teach it how an internal combustion engine works. You have to teach it how to accelerate, brake, steer, and shift gears, by working those controls.

So now let's relate this back to a programming example.

Twitter has an API. That means you can write a program that accesses or controls Twitter. You don't have to know how Twitter works internally. You just have to know their **API interface** for programmers to interact with their service. Most of the major web services like Google, Facebook, and Reddit … all have APIs you can use to automate almost anything you would do manually.

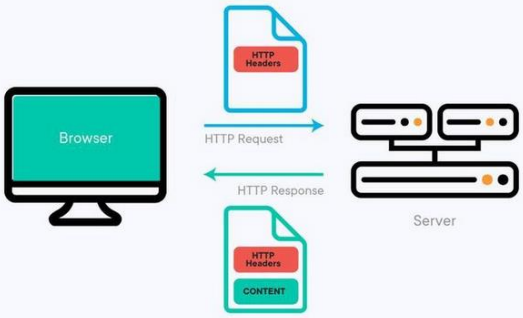


Application Programme Interface abbreviated as API which software intermediary that allows two applications to talk to each other.Let's look into an example to go through with this topic,

01 If user wants to book train tickets, web application shows the departure time, stations like that all the informations.

02 When we look in this from the developer side, developer wants to display all the train schedule informations in the web application, then developer use the APIs to get these train informations from the web server where all the train details and informations are stored.

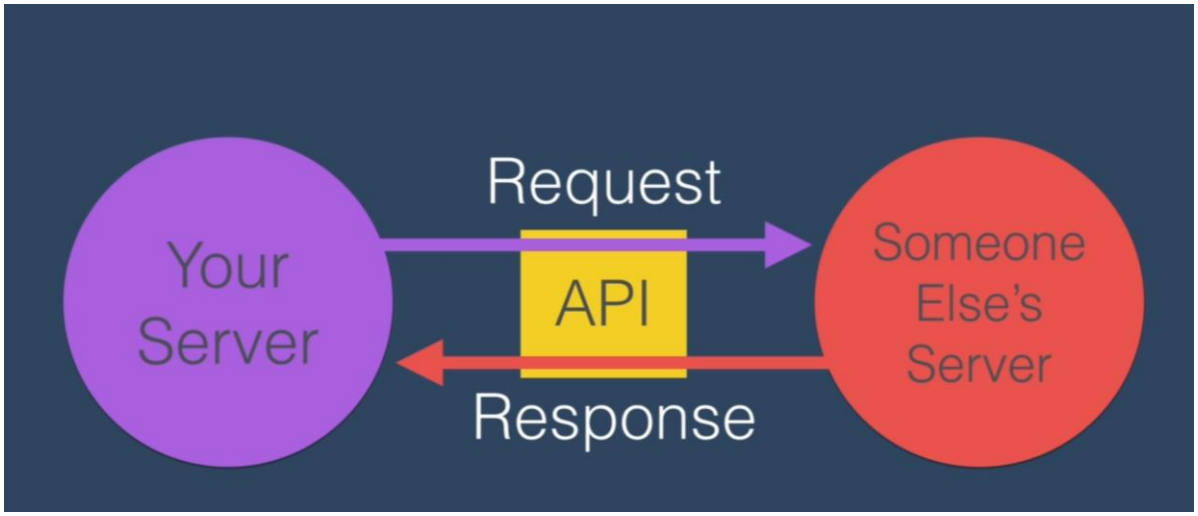
03 Then developers can retrieve data from the web server using the APIs to display in the web application.



There **four basic HTTP requests** a client can make are:

- **GET** — To retrieve a resource.
- **POST** — To create a new resource.
- **PUT** — To edit or update an existing resource.
- **DELETE**- To delete a resource.

This is what API does in our web development.



Four things need to discuss in API:

- Endpoints -the URL of API -Example Kanya west jokes API.
- Paths-Example -Jokes API
- Parameters-**For custom Queries** -the companies give parameter in key: value pair. After the user "?" through the use of paths and parameters, we're able to narrow down the data.
- Authentication-Example open weather:

  https://samples.openweathermap.org/data/2.5/weather?q=London,uk&appid={}

  Paste your own API id in {}

Challenge:

Change the temperature to another unit. Sometimes changing URLs and adding parameters are quite fiddly so for creating and testing API, we use a software called POSTMAN. (Use browser version)

JSON

JSON stands for JavaScript Object Notation
JSON is a lightweight format for storing and transporting data.
JSON is often used when data is sent from a server to a web page.
JSON is "self-describing" and easy to understand.

JSON Syntax Rules
* Data is in name/value pairs
* Data is separated by commas
* Curly braces hold objects
* Square brackets hold arrays - because it's basically the same as JavaScript objects (and arrays)!

We can save data in json files (e.g. weather.json).

Weather.json


```
{
  "coord": {
    "lon": -0.1257,
    "lat": 51.5085
  },
  "weather": [
    {
      "id": 804,
      "main": "Clouds",
      "description": "overcast clouds",
      "icon": "04d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 18.51,
    "feels_like": 17.7,
    "temp_min": 16.72,
    "temp_max": 19.49,
    "pressure": 1014,
    "humidity": 49
  },
  "visibility": 10000,
  "wind": {
```

```
    "speed": 5.14,
    "deg": 200,
    "gust": 10.8
  },
  "clouds": {
    "all": 98
  },
  "dt": 1656269816,
  "sys": {
    "type": 2,
    "id": 2019646,
    "country": "GB",
    "sunrise": 1656215078,
    "sunset": 1656274909
  },
  "timezone": 3600,
  "id": 2643743,
  "name": "London",
  "cod": 200
}
```


Tip:Install JSON awesome viewer in chrome for pretty view.

Open index.js. Create a JSON

```
let person = {
    "name": "Amira",
    "pets": [
        {
            "species": "cat",
            "name": "Popcorn"
        },
        {
            "species": "cat",
            "name": "Pumpkin"
        }],
    "favouriteFoods": ["jacket potato with cheese and butter", "watermelon",
"strawberries dipped in chocolate"]
}

console.log(person);
```

Exercise 1:

Make an object in JavaScript, it should contain:

- your name
- your age
- your contact details
    - your phone number
    - your email address
- your physical address
    - street name/house number/house name
    - city
    - country
    - postcode
    - county

Part 1: console.log your contact details object console.log your name and age

fetch

We can make a request on page load (in the script tag and the link tag etc) for data or resources. Sometimes we need to make a request after the page has loaded. This might be:

- to render the page first, so you can show a loading bar while you wait for a large amount of data that will take longer to load
- to update the data already on the page, such as:
    - to filter products in a shop for a specific size
    - to regularly update the number of tickets that are left for a popular concert
- to get new data after a user does something
    - to get a product description when someone clicks on an item in a list of products

For this we use a function called fetch().

Syntax:

```
fetch("example.json") // call a filepath or url
```

Fetching data can take a while, so this function doesn't return data in the normal way. We call a method called .then() on the fetch, because it might take a while for the fetch request to finish, and this will wait for us. It will take a callback, and pass you the response:

```
fetch("example.json")
    .then(response => {
      // response is in here
      // but you don't have access to the JSON data
    })
```

When you get the response, you need to access the data. Do that by calling response.json() and returning that value.

This will unpack the JSON, which might also take a while, so you need you to call .then() again. Like this:

```
fetch("example.json")
    .then(response => {
        return response.json()
    }).then(data => {
        // here you have access to the JSON data
        // data is JSON - i.e. a JavaScript object
    })
```

.catch()

.catch() catches errors. If anything goes wrong at any point, it will run a callback. Call it in the same way you call .then().

```
fetch("example.json")
    .then(response => {...})
    .then(data => {...})
    .catch((error) => {
        // handle error in here
      console.log(error)
    })
```

You can also throw errors yourself (e.g. if the data is invalid), and it will run the catch. It will ONLY run if there's an error.

```
fetch("example.json")
    .then(response => {
        return response.json()
    })
    .then(data => {
        if (!isArray(data)) {
            throw new Error("Invalid response: data is not an array.")
        }
    })
    .catch((error) => {
      console.log(error) // Error: Invalid response: data is not an array.
    })
```

Using this practically ourselves

To use fetch your code must be running through a server. There are many ways to achieve, the easiest way at this point is to let our IDE do it for us.

If you're running a server, your URL will look a bit like this:

http://localhost:63342/url-or-file-path

- Localhost is where all local servers are run.
- The number after the colon is a "port", and that just tells us which part of the computer is listening to handle requests

Exercise 2

Part 1:

- Make a fetch request to the json file I shared in Slack. It has a collection of people.(example.json)
- console.log the JSON inside it.

Part 2:

Display each person in your HTML.

Part 3 (stretch):

Make a button that will change your HTML to only show the people with pets.