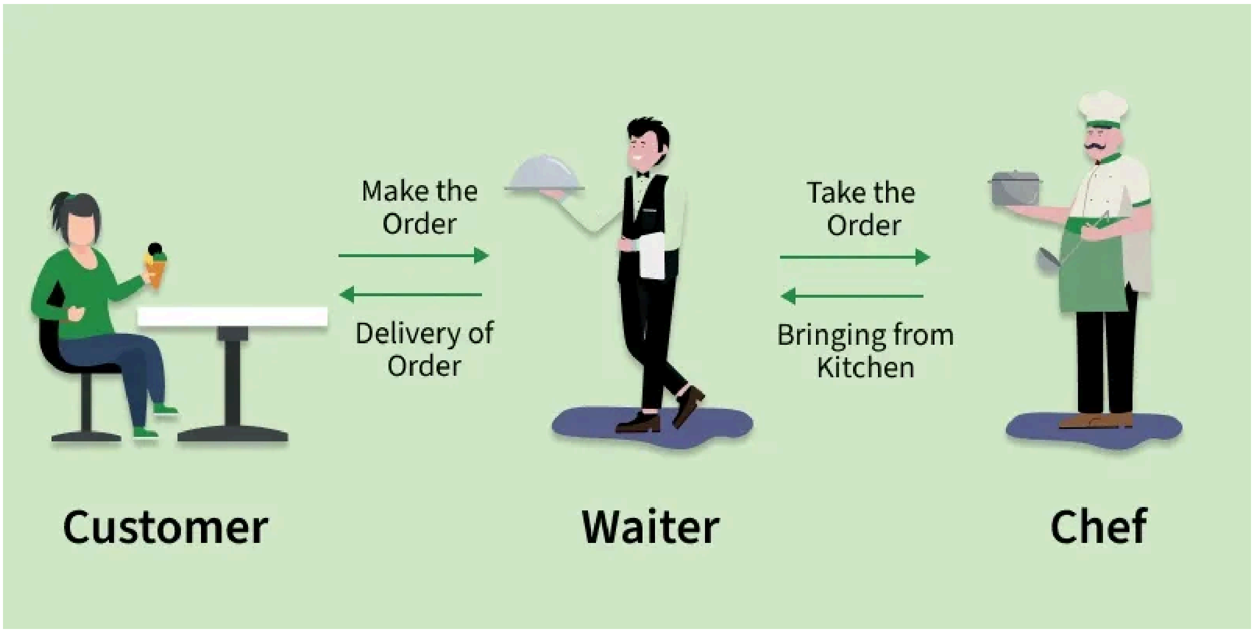
API (Application Programming Interface)

Connecting client and server. An API is a set of rules that allows one software application to interact with another.



Example: If you want to show weather updates in your app, instead of building a weather system, you can simply use the OpenWeatherMap API to fetch the data instantly.

An API:

- Defines **what requests you can make**
- Defines **what data you will get back**
- **Hides** internal complexity

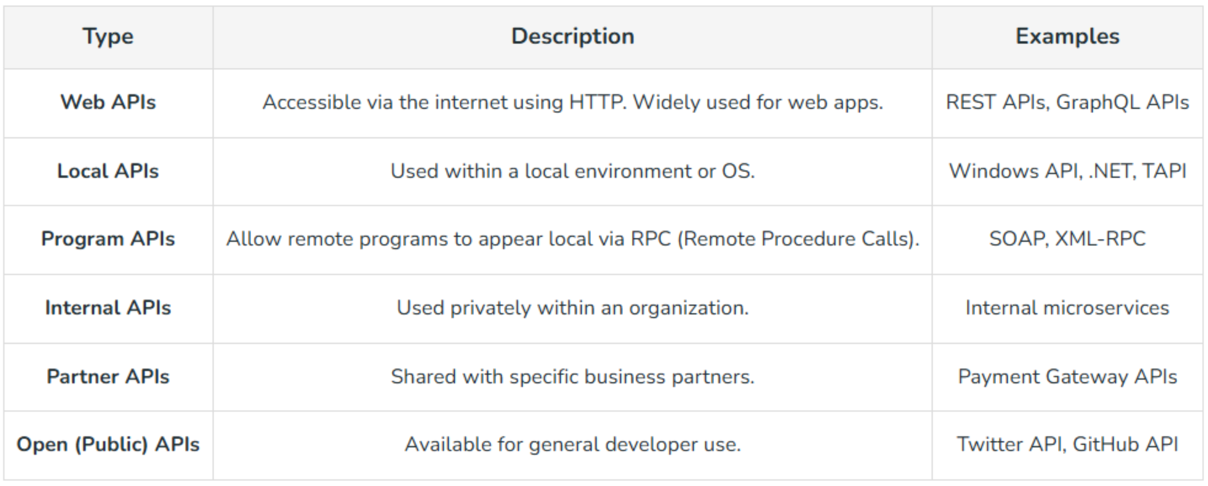
How Do APIs Work?

APIs operate through a request response cycle between a client and a server

- Request: The client sends a request to an API endpoint (URI).
- Processing: The API forwards the request to the server.
- Response: The server processes and sends back the requested data.
- Delivery: The API returns the server's response to the client.

This communication happens over the HTTP/HTTPS protocol, with additional security via headers, tokens, or cookies.

Types:



Type	Description	Examples
Web APIs	Accessible via the internet using HTTP. Widely used for web apps.	REST APIs, GraphQL APIs
Local APIs	Used within a local environment or OS.	Windows API, .NET, TAPI
Program APIs	Allow remote programs to appear local via RPC (Remote Procedure Calls).	SOAP, XML-RPC
Internal APIs	Used privately within an organization.	Internal microservices
Partner APIs	Shared with specific business partners.	Payment Gateway APIs
Open (Public) APIs	Available for general developer use.	Twitter API, GitHub API

REST APIs

A REST API is an application programming interface (API) that conforms to the design principles of the representational state transfer (REST) architectural style. In REST Architecture everything is a **resource**. It's more of an approach that's centered around resources and things you can do to resources. The HTTP verbs GET, POST, PUT and DELETE are typical actions that you can apply against any resource.

Some APIs, such as SOAP or XML-RPC, impose a strict framework on developers. But developers can develop REST APIs using virtually any programming language and support a variety of data formats. The only requirement is that they align to the following six REST design principles, also known as architectural constraints.

Architectural Constraints of RESTful API

Uniform interface

All API requests for the same resource should look the same, no matter where the request comes from. The REST API should ensure that the same piece of data, such as the name or email address of a user, belongs to only one uniform resource identifier (URI). Resources shouldn't be too large but should contain every piece of information that the client might need.

A REST API must use a uniform interface, meaning each resource has a single, consistent URI, requests look the same across clients, and responses are neither too large nor missing required information.

Client-server decoupling
In REST API design, **client and server applications must be completely independent of each other**. The only information that the client application should know is the URI of the requested resource; it can't interact with the server application in any other ways. Similarly, a server application shouldn't modify the client application other than passing it to the requested data via HTTP.

Statelessness
REST APIs are stateless, **meaning that each request needs to include all the information necessary for processing it**. In other words, REST APIs do not require any server-side sessions. Server applications aren't allowed to store any data related to a client request.

Cacheability
Every response should include whether the response is cacheable or not and for how much duration responses can be cached at the client side. Client will return the data from its cache for any subsequent request and there would be no need to send the request again to the server. Server responses also need to contain information about whether caching is allowed for the delivered resource. The goal is to improve performance on the client side, while increasing scalability on the server side.

Layered system architecture
In REST APIs, the calls and responses go through different layers. As a rule of thumb, **don't assume that the client, and server applications connect directly to each other.** There may be a number of different intermediaries in the communication loop. **REST APIs need to be designed so that neither the client nor the server can tell whether it communicates with the end application or an intermediary.**

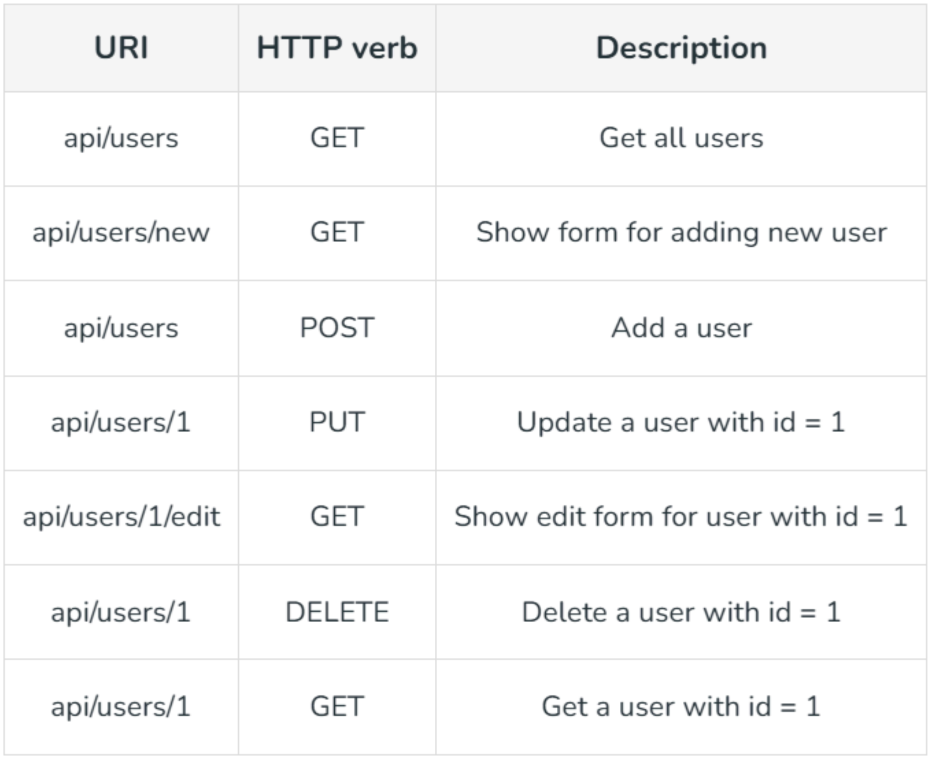
Code on demand (optional)
The server sends code instead of (or along with) data.
REST APIs usually send **static** resources, but in certain cases, responses can also contain executable code (such as Java applets). In these cases, the code should only run on-demand.

bandwidth: REST does not need much bandwidth when requests are sent to the server. REST messages mostly just consist of JSON messages.

Rules of REST API

1. Noun endpoint / Resource. It means that a URI of a REST API should always end with a noun. Example: /api/users is a good example
2. Request using HTTP Action Verb (GET, POST etc)

3. Plural of resources (/users, /users/1, not user/1).
4. HTTP Response Code
5. A RESTful web service must be very well organized (for example, when I see DELETE /api/users/1, I understand that the user with ID 1 needs to be deleted). And as a developer it should be clear that what needs to be done just by looking at the endpoint and HTTP method used.



URI	HTTP verb	Description
api/users	GET	Get all users
api/users/new	GET	Show form for adding new user
api/users	POST	Add a user
api/users/1	PUT	Update a user with id = 1
api/users/1/edit	GET	Show edit form for user with id = 1
api/users/1	DELETE	Delete a user with id = 1
api/users/1	GET	Get a user with id = 1

```
router.get("/request", (req,res)=>{
        res.send("<h1>This is a GET request</h1>")
});
```

```
localhost:8080/api/request
```

With parameters:

```
router.get('/users/:id', (req, res) => {
          const userId = req.params.id;
          // Now you can use the userId in your code
          res.send(`User ID: ${userId}`);
});
```

```
localhost:8080/api/users/123
```

Asynchronous Flow

Control Flow is the order in which statements are executed in a program. By default, JavaScript runs code from top to bottom and left to right. Control flow statements let you change that order, based on **conditions, loops or keywords**.

JavaScript runs on a **single thread**. **It can only do one thing at a time**. Every task has to wait for the previous one to finish. This can freeze an application during slow operations (like file requests).

Synchronous vs Asynchronous

Synchronous: Tasks run one after another, blocking execution
Asynchronous: Long-running tasks run in the background, allowing the program to continue. Functions running in **parallel** with other functions are called asynchronous

Why Async is Needed

- Network requests (API calls)
- File reading
- Timers
- User interactions

JavaScript is single-threaded, so async patterns prevent the UI from freezing.

Custom Events

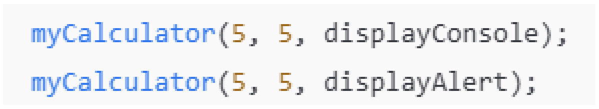
https://codepen.io/FARZANA-TABASSUM-Lecturer-CSE/pen/qENWGjr

Callbacks

https://codepen.io/FARZANA-TABASSUM-Lecturer-CSE/pen/ByzBeVM?editors=0010
A callback is a function passed as an argument to another function.

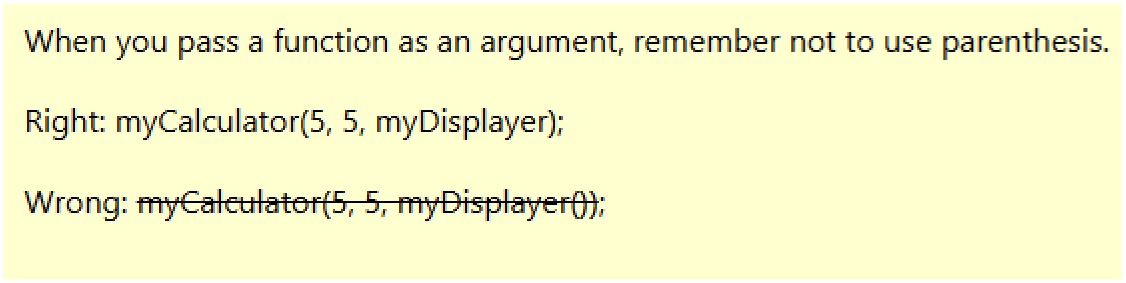
What's the REAL difference

- ➔ Without callback: myCalculator is hard-coded to use myDisplayer.
    - ◆ It decides how the result will be used
    - ◆ You cannot change the behavior without modifying myCalculator
    - ◆ This is called tight coupling.

- With callback: myCalculator does only one job: calculate.
    - It does NOT care what happens next
    - The caller decides how to use the result
    - This is loose coupling.
    - 

```
myCalculator(5, 5, displayConsole);
myCalculator(5, 5, displayAlert);
```

- Asynchronous (main point): setTimeout



When you pass a function as an argument, remember not to use parenthesis.

Right: myCalculator(5, 5, myDisplayer);

Wrong: ~~myCalculator(5, 5, myDisplayer());~~

Where callbacks really shine are in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

Promises

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

- A Promise is a **proxy** for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's **eventual success value or failure reason**. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

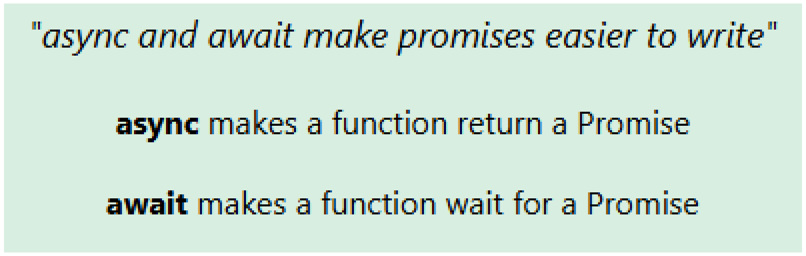
Promise States
Pending – initial state
Resolve – operation successful
Rejected – operation failed

https://codepen.io/FARZANA-TABASSUM-Lecturer-CSE/pen/RNRbzPE?editors=0011

How Promises avoid race conditions?
Codepen example

Async/Await

https://codepen.io/FARZANA-TABASSUM-Lecturer-CSE/pen/VYjZJbq

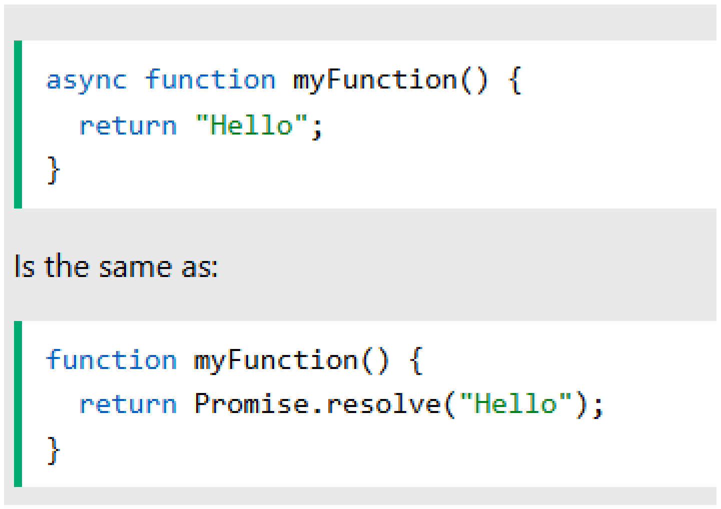


"async and await make promises easier to write"

async makes a function return a Promise

await makes a function wait for a Promise

Example:



```
async function myFunction() {
  return "Hello";
}
```

Is the same as:

```
function myFunction() {
  return Promise.resolve("Hello");
}
```

Routes

Node.js: The JavaScript Runtime Environment
Node.js is an open-source, cross-platform runtime environment that allows developers to execute JavaScript code outside of a web browser.

Express.js: The Web Application Framework
Express.js is a minimalist, flexible, and popular framework built on top of Node.js

A route is a URL path that maps an incoming request to a specific action or resource on the server. A route tells the server what to do when a particular URL is requested.

Route Parameters:

/users/:id	GET /users/42

```
app.get("/users/:id", (req, res) => {
  console.log(req.params.id);
});
```

Query Parameters

/users?role=admin&active=true	GET /products?category=phone&sort=price

```
app.get("/products", (req, res) => {
  console.log(req.query.category);
  console.log(req.query.sort);
});
```

Code example