Embrace Opportunity

Software Engineering Day 6

Asynchronous JavaScript and Introductions to APIs and Full Stack Development

i_X

Today's Overview

- Asynchronous Javascript.
- Javascript event loop.
- Call back functions, promises, async/await.
- Introductions to Backends, APIs and REST.

- Cheat Sheet:
    - iX Cheat Sheet
    - iX Cheat Sheet - Day 6

JavaScript

Asynchronous Javascript

Asynchronous JavaScript

- Single threaded, non-blocking, asynchronous, high level object oriented programming language (abstraction)
- Interpreted (just-in-time compiling)
- ECMAScript specification (ES14)
- Asynchronous operation will execute another block while waiting for another operation to complete.
- JavaScript relies on callback queue and event loops for asynchronous operations.

Asynchronous JavaScript

- Asynchronous JavaScript utilizes three techniques:
- Older:
  - *Promises*
    - They run blocks of code after a defined period of time or when a defined condition is satisfied.
  - *Callbacks*
    - Allows asynchronous blocks to be written in a synchronous manner.
    - Function that executes after the outer code call has finished, by passing a function to another function as an argument.
- Newer [Most Utilized]:
  - *Async/Await*
    - More recent addition to JavaScript which gives syntactic sugar for asynchronous code compared to *Promises*.

JavaScript

Event Loop

Event Loop - Introduction

- Non-blocking programming model:
    - Program executes instructions independently of each other, without waiting for one to complete before moving on to the next. When a non-blocking operation is encountered, the program initiates the operation and then continues to execute other instructions.
        - Once the operation is completed, a callback function is called to handle the result.
    - Purpose:
        - Improves responsiveness, concurrency and efficiency compared to blocking programming.

Event Loop - Introduction

- Event loop and JS runtime:
  - JavaScript has a runtime model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks.
- The event loop is the orchestrator that ties the Call Stack, Callback Queue, and Web APIs together.
  - Its primary role is to constantly check whether the call stack is empty.
- Event loop sequence:
  - Checking call stack
  - If the call stack is empty - Move function from callback queue to call stack
  - If not - Keep checking the call stack

Event Loop - Introduction

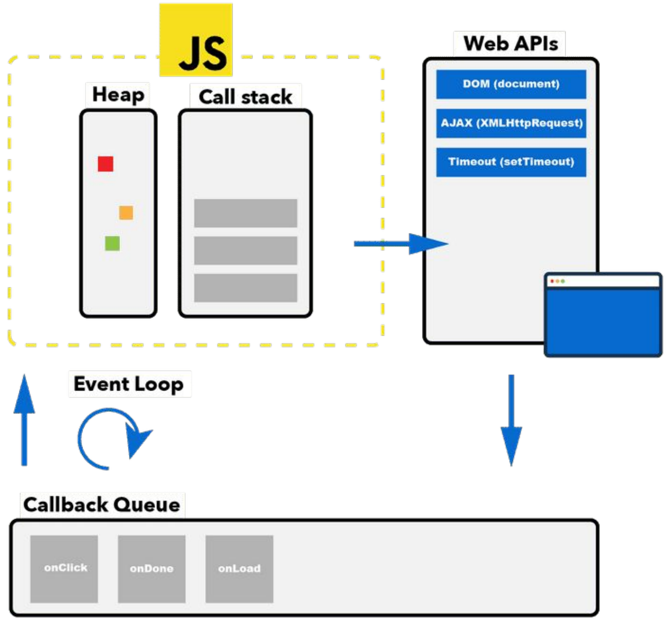
- JS engine consists of three main components:
    - Memory Heap
        - Where the memory allocation happens, all object variables are assigned here in a random manner.
    - Call Stack
        - Where function calls are stored.
        - LIFO (last in first out) data structure.
    - Callback Queue
        - When asynchronous operations are complete, corresponding functions (callbacks) are placed in the callback queue.

Event Loop - Overview

- Summary:
  - Call Stack
    - Keeps track of the currently executing functions.
    - When a function is finished it is popped from the stack.
  - Callback Queue
    - When asynchronous operations are complete, corresponding functions (callbacks) are placed in the callback queue.
  - Event Loop
    - Continuously checks the call stack and the callback queue.
    - If the call stack is empty, it takes the first function from the callback queue and pushes it on the call stack for execution.

Event Loop – Runtime Concept

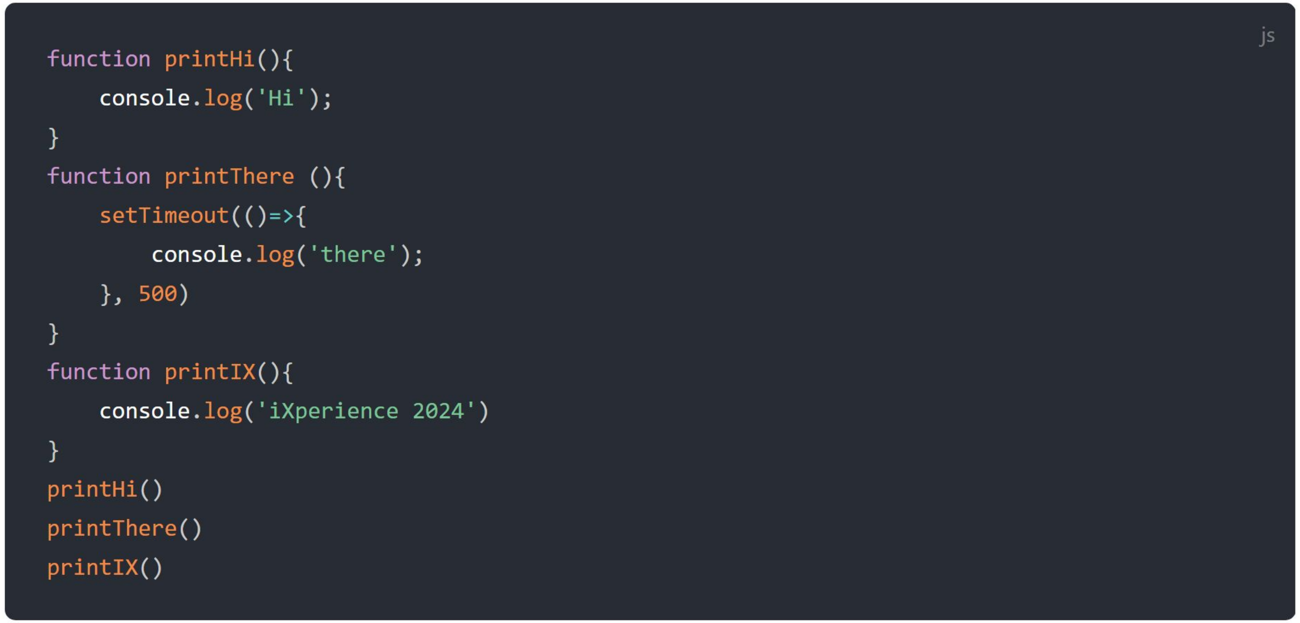
- Event loop model broken into:
  - *Stack*
    - Function calls from a stack of frames.
  - *Heap*
    - Objects are allocated in a heap which is large region of memory.
  - *Queue*
    - List of messages to be processed.
    - Each message has an associated function that gets called to handle the message.



iX

Event Loop - Visualized

- Let us look at two examples of the event loop using the code snippet below, with slight changes to the *timeout* to see how it affects the results.



js

```js
function printHi(){
    console.log('Hi');
}
function printThere (){
    setTimeout(()=>{
        console.log('there');
    }, 500)
}
function printIX(){
    console.log('iXperience 2024')
}
printHi()
printThere()
printIX()
```

Event Loop - Visualized

- Example two, notice the *setTimeout*.

```js
function printHi(){
    console.log('Hi');
}
function printThere (){
    setTimeout(()=>{
        console.log('there');
    }, 0)
}
function printIX(){
    console.log('iXperience 2024')
}
printHi()
printThere()
printIX()
```

JavaScript

Call back functions, promises, async/await

Callback - Introduction

- A callback is just a function that’s passed into another function, with the expectation that the callback will be called at the appropriate time.
  - Callback is a function that executes after the outer code call has finished running.
- Having to call callbacks inside callbacks (nested callbacks) makes it very difficult to both read and handle errors at each callback.
  - Also known as “callback hell”.
- That is why asynchronous JavaScript utilizes:
  - *Promise*
  - *Async/Await*

Callback - Utilizing Callback - Example

```
fetchBlogs(
    (res)=>{
        setBlogs(res);
    }
);

const fetchBlogs = async (cb) => {
    fetch("http://localhost:8000/api/blogs", {
        method: "GET",
        headers: {
            "Content-Type": "application/json",
        },
    }).then(res => {
        cb(res);
    })
}
```

Promise - Introduction

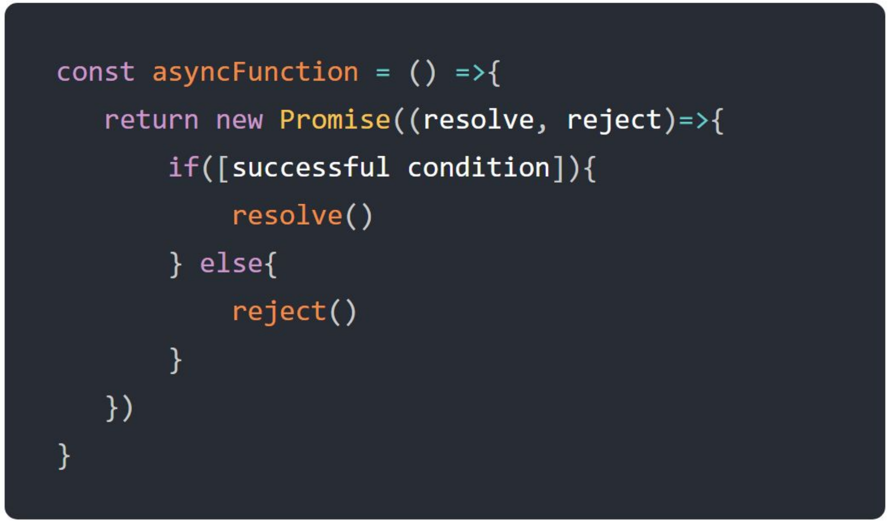
- A promise is an object returned by an asynchronous function, which represents the current state of the operation.
- Promise object provides methods to handle the eventual success or failure of an operation.
    - This is due to times when the promise is returned to the caller but the operation isn’t finished at that moment.

Promise - Introduction

- Promises three states:
    - *Pending*:
        - The initial state; neither fulfilled nor rejected.
    - *Fulfilled*:
        - Operation completed successfully, and the Promise has a resulting value.
    - *Rejected*:
        - Operation failed, and the Promise has a reason for the failure.
- Promise is closed when it is either *fulfilled* or *rejected*.

Promise - Creation

- Creating a *Promise*:
  - *Promise* constructor - Single argument
    - Executor - Double argument
      - Resolver function
        - Called on success resolution
      - Reject function
        - Called on failure resolution



```
const asyncFunction = () =>{
   return new Promise((resolve, reject)=>{
       if([successful condition]){
           resolve()
       } else{
           reject()
       }
   })
}
```

Promise - Handling Result

- To handle the results of the *Promise*, there are three callback methods that can be called when the state of the *Promise* has changed:
  - .then()
    - Callback that is called when the Promise state is fulfilled.
  - .catch()
    - Callback that is called when the Promise state is rejected.
  - .finally()
    - Callback that is called when the Promise state is settled.
    - Considers any outcome

Promise - Handling Result

- On Success:

```
myPromise.then((value) => {console.log('Promise fulfilled with value: ', value); });
```

- On Failure:

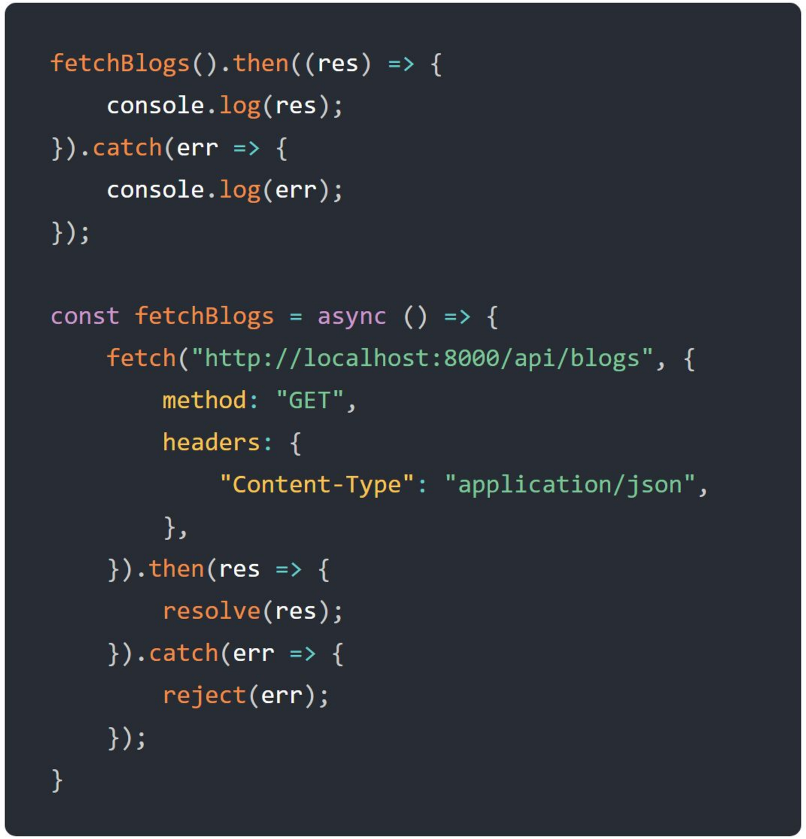
```
myPromise.catch((error) => {console.log('Promise rejected with reason: ', error.message); });
```

- Regardless of Outcome (Settled = Success or Failure):

```
myPromise.finally(() => {console.log('Promise settled'); });
```

Promise - Chaining Promises

- Promises can be chained together using the *.then()* method.
- .then() itself returns a promise, which will be completed with the result of the function passed to it.



```
fetchBlogs().then((res) => {
    console.log(res);
}).catch(err => {
    console.log(err);
});

const fetchBlogs = async () => {
    fetch("http://localhost:8000/api/blogs", {
        method: "GET",
        headers: {
            "Content-Type": "application/json",
        },
    }).then(res => {
        resolve(res);
    }).catch(err => {
        reject(err);
    });
}
```

Async/Await - Introduction

- Recent addition to JavaScript, providing syntactic sugar for handling asynchronous code using Promises.
- *async* and *await*, allow you to write functions that behave like synchronous functions.
    - async
        - Called before the function.
        - Always returns a Promise.
    - await
        - Called inside an *async* function.
        - Halts execution till the Promise is fulfilled or rejected.
- Used commonly when fetching from an API.

Async/Await - Syntax

- Function Syntax:

```
async function myFunction() {
    const response = await fetch(/* ENDPOINT */);
    const responseData = await response.json();
    return responseData;
}
```

- Arrow Function Syntax:

```
const myFunction = async () => {
    const response = await fetch(/* ENDPOINT */);
    const responseData = await response.json();
    return responseData;
}
```

Async/Await - Example

```javascript
const fetchCategories = async () => {
  try {
    const response = await fetch("http://localhost:8000/api/categories");
    let res = await response.json();
    console.log("Data retrieved successfully: ", res);
  } catch (error) {
    console.log("Error occurred");
  }
};
```

Asynchronism - Error Handling

- Error handling can be used with the:
- *try…catch()*:

```
const myFunction = async () => {
    try {
        //Attempt code block
        const response = await fetch("ENDPOINT");
        const responseData = await response.json();
        return responseData;
    } catch(error) {
        // On failure
        console.log(error.message);
    }
}
```

Asynchronism - Error Handling

- Promises can use the *then()...catch()* methods for error handling:

```
myPromise
    .then((response) => {
        // On Success
        setData(response); //Utilizing the response
    })
    .catch((error) => {
        // On Failure
        throw new Error(error.message); //Throwing an error
        setErrorMessage(error.message) //Or setting an error message
    });
```

Backend Server

The **heart** of a web application

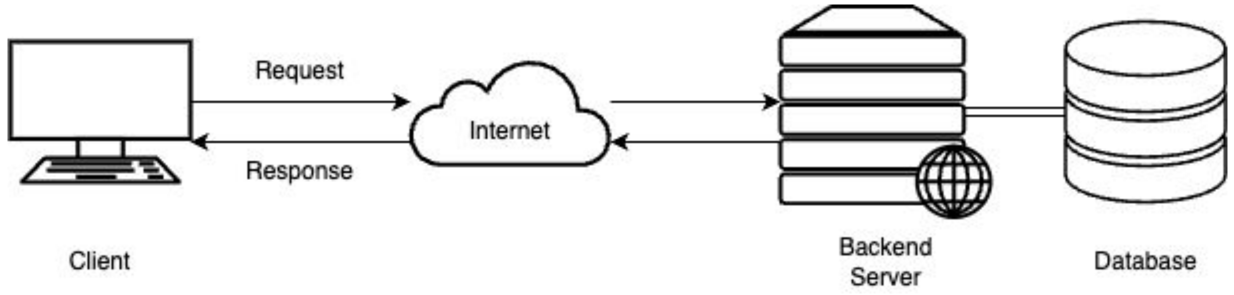
Definition of a Backend

The backend of a web application is the **server-side**, where all the data processing happens. It's responsible for storing, retrieving, and manipulating data. The backend of a web application handles business logic, database interactions, authentication, and authorization.

In **Full Stack** development, the backend is the server-side of the client-server communication.

Definition of a Web Server

A **web server** consists of both software and hardware components designed to deliver web content and services to clients via the internet. It hosts and manages web applications, providing access to application interfaces and RESTful web services. This setup enables web browsers and other applications to interact with hosted applications, facilitating data exchange and dynamic content delivery.



API

Application Programming Interface

Definition of an API

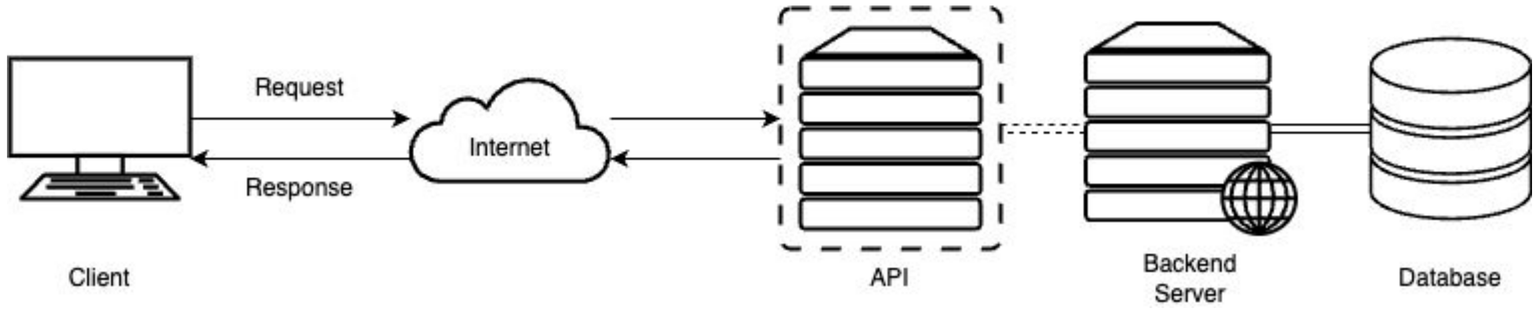
An API, or Application Programming Interface, is a **set of functions and protocols** allowing applications to access the features and/or data of an operating system, application, or other service.

Essentially, it's a way for different programs to interact with each other.

How APIs work

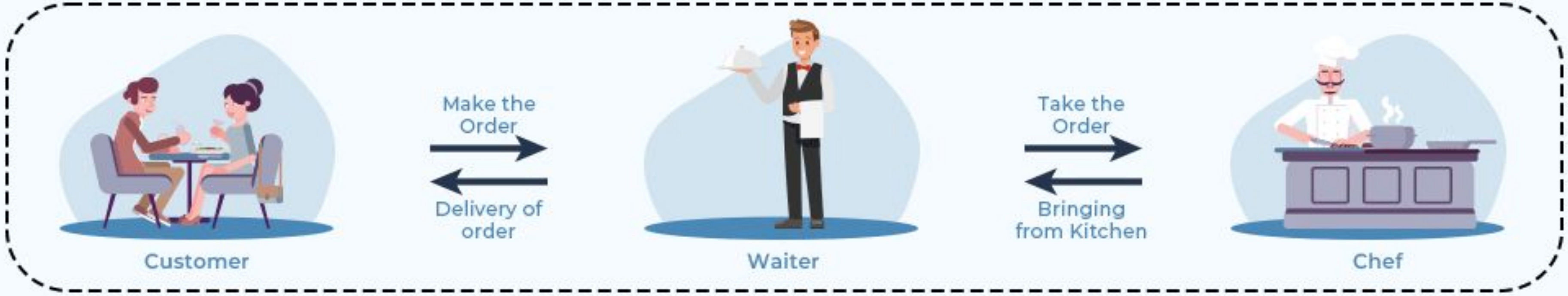
An API is an abstraction layer to the underlying implementation by only exposing objects or actions.

An example of an API is an intermediary layer that enables multiple frontend applications to send and receive data to and from a backend or web server.

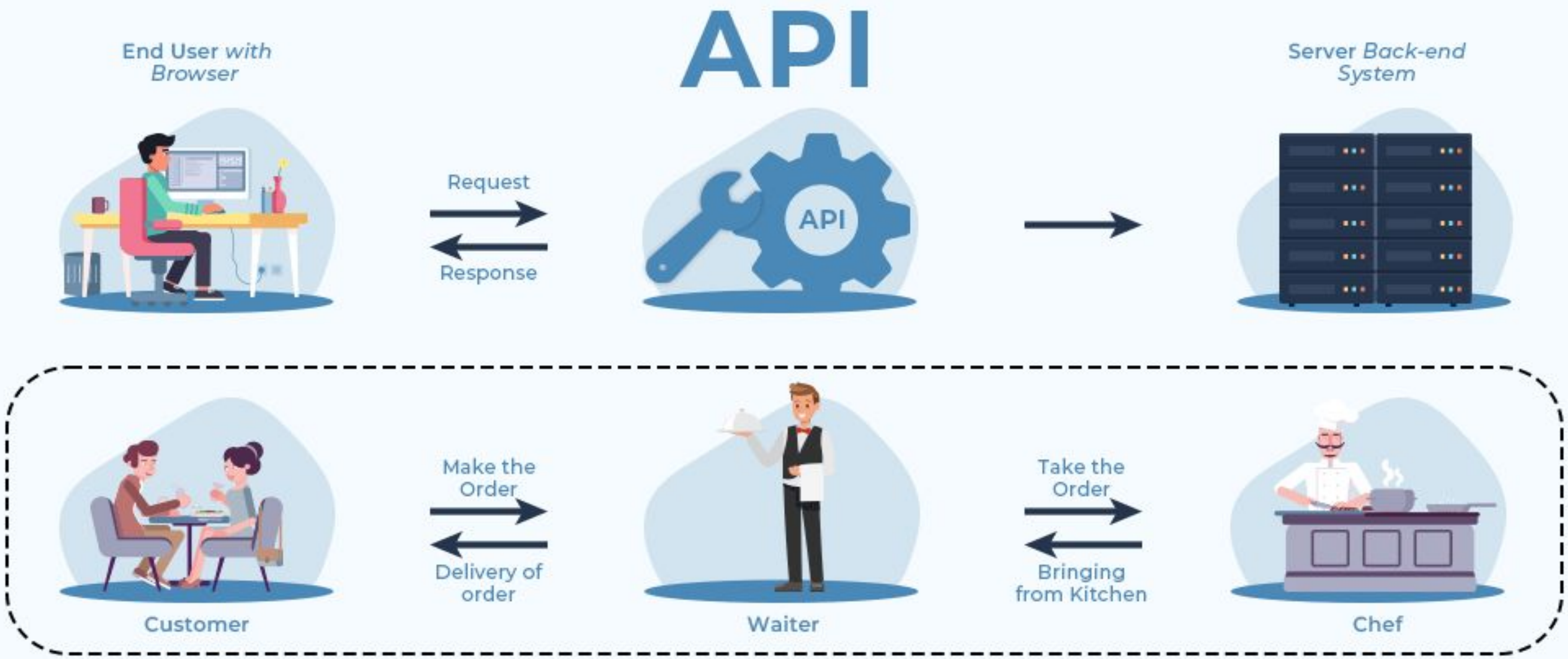


API analogy

Imagine you're sitting at a table in a restaurant with a menu of choices to order from. The kitchen is the part of the “Backend” that will prepare your order, and you are the part of the “Frontend”. **An API can be thought of as a waiter** in the restaurant, to communicate your order to the kitchen and deliver your food back to your table.



API analogy


API
End User with Browser
Request
Response
API
Server Back-end System
Make the Order
Delivery of order
Take the Order
Bringing from Kitchen
Customer
Waiter
Chef

REST

Representational State Transfer

REST - Introduction

- What is REST
  - REST, or Representational State Transfer, is an architectural style that defines a set of constraints to be used for creating web services. REST APIs enable client-server communication in a standardized way, making web development more efficient and scalable.

- What is a RESTful API:
  - RESTful APIs facilitate communication between different parts of a system in a standardized way, enhancing overall system architecture and performance.

REST - Key principles

- **Client-Server Architecture:** The client and the server should act independently. They interact with each other through requests (from the client) and responses (from the server).
- **Stateless:** As mentioned, no client context is stored on the server between requests. Each request is treated as new.
- **Cacheable:** Clients can cache responses. Responses must, therefore, implicitly or explicitly label themselves as cacheable or not, to prevent clients from reusing stale or inappropriate data in response to further requests.
- **Layered System:** A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.

REST - Key principles

- **Uniform Interface:** To obtain the uniformity throughout the application, REST has defined four interface constraints – Resource-Based, Manipulation of Resources Through Representations, Self-descriptive Messages, and Hypermedia as the Engine of Application State (HATEOAS).
- **Code on Demand (optional):** Servers can temporarily extend or customize the functionality of a client by transferring executable code.

https://searchmicroservices.techtarget.com/definition/REST-representational-state-transfer

https://martinfowler.com/articles/richardsonMaturityModel.html

HTTP

Hypertext Transfer Protocol

What is HTTP?

- HTTP stands for **Hypertext Transfer Protocol**. It's the foundation of data communication on the World Wide Web.
- HTTP allows for the **fetching of resources**, such as HTML documents. It is a protocol that clients (usually web browsers) use to request resources from servers.
- **REST uses HTTP** as the underlying communication protocol.
- HTTP is **stateless**
- HTTPS is **secure**, by encrypting data traffic

HTTP methods

HTTP methods are a critical component of the HTTP protocol, defining the action to be performed on a given resource. They indicate the desired action (such as fetching, creating, updating, or deleting data) to be applied to a resource identified by a given request URI (Uniform Resource Identifier). Below are the most commonly used HTTP methods, each serving a specific purpose in the context of RESTful APIs:

- **GET:** Used to request data from a specified resource. GET requests should only retrieve data and have no other effect.
- **POST:** Used to send data to a server to create a new resource. POST requests are often used to submit form data or upload a file.
- **PUT:** Used to send data to a server to update an existing resource or create a new resource if it does not exist. PUT requests replace the current representation of the target resource with the request payload.
- **DELETE:** Used to delete the specified resource.

HTTP methods

- **HEAD:** Similar to GET, but it retrieves only the headers of a resource, which means it is used for getting the metadata of the resource.
- **OPTIONS:** Describes the communication options for the target resource, allowing clients to determine the capabilities of the web server.
- **PATCH:** Used to apply partial modifications to a resource, contrasting with PUT, which fully replaces the resource.

HTTP response codes

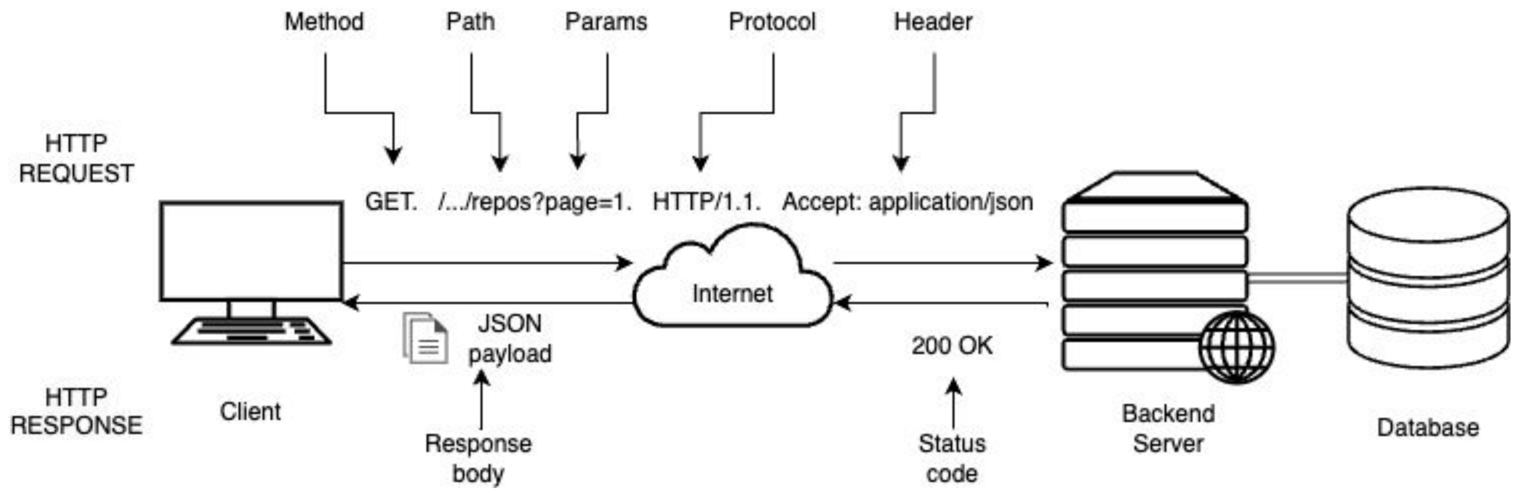
Common HTTP Status Codes

- **1xx Informational**
  - **100 Continue:** The client should continue with the request or ignore if it is already finished.
- **2xx Success**
  - **200 OK:** The request has succeeded.
  - **201 Created:** The request has been fulfilled, resulting in the creation of a new resource.
- **3xx Redirection**
  - **301 Moved Permanently:** The requested resource has been assigned a new permanent URI.
  - **304 Not Modified:** Indicates that the resource has not been modified since the version specified by the request headers.

HTTP response codes

- 4xx Client Errors
  - **400 Bad Request:** The server cannot process the request due to a client error.
  - **404 Not Found:** The requested resource could not be found.
- 5xx Server Errors
  - **500 Internal Server Error:** A generic error message when an unexpected condition was encountered.
  - **503 Service Unavailable:** The server is currently unable to handle the request due to temporary overloading or maintenance.

Anatomy of an HTTP request to RESTful API



Homework

Apply what we have learned

Homework

- Updating blog app frontend to fetch data from the teaching teams API.
- Requirements:
  - Fetching data from the API
    - *https://ix-blog-app-2d5c689132cd.herokuapp.com/*
  - Fetching:
    - Blog Posts:
      - All blogs - *fetchBlogs*
        - *https://ix-blog-app-2d5c689132cd.herokuapp.com/api/blogs*
      - Blogs by category ID - *fetchBlogsByCategoryId*
        - *https://ix-blog-app-2d5c689132cd.herokuapp.com/api/blogs/category/[category-id]*

Homework

        - Categories:
            - All categories - *fetchCategories*
                - *https://ix-blog-app-2d5c689132cd.herokuapp.com/api/categories*
- Updating blogs and categories with the fetched data rather than the imported dummy data.
    - *blogs*
    - *categories*

Next Class

Introduction to Node.js and Express