# Basics

Synchronous programming, we need to wit until pervious tasks are over.

ret = getRemoteData();

// wait till remote data is fetched

// process returns from remote data

// next operation has to wait until

// all previous steps are done

nextOperation();

In async programming we don’t wait for blocking operation to finish, instead we register a function (callback) to be called when blocking operation is done and we continue executing another code.

getRemoteData(function(){

// wait till remote data is fetched

// process return from remote data

});

// next operation don’t have to wait

// for old operation to finish

nextOperaiton();

I JavaScript most of Async implementations are supported by different libraries.

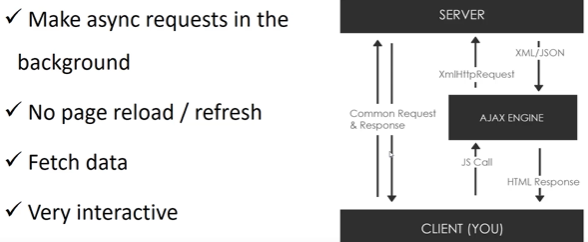
* XMLHTTPRequest & Fetch
* JQuery, Ajax, Axios, other HTTP libraries
* Node.js fs (filesystem) module

Following are ways we can handle Asynchronous code

* Callbacks
* Promises
* Async/Await

# AJAX

Asynchronous JavaScript and XML. AJAX  **is a buzzard for ‘making a HTTP request from Javascript without leaving or refreshing page’** . This is not any single library of framework, but set of web technologies to send and receive data between client and server. In Ajax mostly JSON is used rather than XML (even if it’s in name). Most of the API’s now return JSON data and not XML.



XMLHTTPRequest (XHR) is one of the (OLD technology now) API used for Ajax. This is API is supported/ is part of all the browser’s. This API is mainly used for transfer of data between client and server and can work with JSON, XML, or plain text.

There are **other newer/external** libraries as well which can help making Ajax call.

* Fetch API (popular and built-in to Javascript)
* Axios (external library)
* Superagent (external library)
* jQuery (not recommended)
* Node HTTP (for nodejs)

## XHR API

Basic syntax of getting data from API endpoint which doesn’t need any authentication

// create XHR object from built-in API

const xhr = new XMLHttpRequest();

// open the connection

xhr.open('GET', <<API URL>>, true);

// this runs when data is returned and loaded

xhr.onload = function(){

  // check if return is success

  if(this.status === 200) {

    console.log(this.responseText);

  }

}

//required to complete the request

xhr.send();

## CallBack function

It is a function passed to another function and it’s called by another function whenever needed (mostly called when another function does its execution)

// if we want 'SayName' function to wait for greet function to finish then

// we can pass 'SayName' function as callback to greet which then internally

// execute call back function

function greet(name, myFunction) {

    console.log('Hello world');

    // callback function

    // executed only after the greet() is executed

    myFunction(name);

  }

  // callback function

  function sayName(name) {

    console.log('Hello' + ' ' + name);

  }

  // calling the function after 2 seconds

  setTimeout(greet, 2000, 'John', sayName);

  //-> 'Hello world' followed by 'Hello John'

## Promises

Introduced in ES6 and they are alternative to callback.

Particular functions when doing async operation it ‘promises’ that it will do something when its operation is done.

Promise can be in following states

* Pending: results are undefined
* Resolved: results are value sent by promise result
* Rejected: error sent my promise results

Function has to return ‘promise’ by creating new object. Promise object accepts a function and provide it default two params ‘resolve’, ‘reject’ as function references.

When resolve(‘ret value’) is called inside function which is passed while creating Promise object then is full fills the promise and if reject(‘ret value’) is called then it rejects the promise.

Resolved promise calls .then() methos and reject methos calls .catch method.

function getData(flag){

    return new Promise(

        function(resolve, reject){

               setTimeout(function(){

                if(flag){resolve('new updated value');}

else{ reject('error in updating value');}

            }, 5000);

        }

    )

}

function setData(val){

    let header = document.querySelector(".container h1");

    header.textContent = val;

}

getData(1)

.then(setData)

.catch(function(err){

    let header = document.querySelector(".container h1");

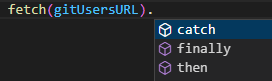
    header.textContent = err;

})

# Fetch API

Fetch API returns promise. We can call API endpoint and wait for fetch promise to be fulfilled in .then method

Fetch support 3 methods .then (fulfilled), .catch(error) and .finally(always run)



**Fulfilment example**

const gitUsersURL = "https://api.github.com/users"

function getGitUsers(){

    // call api in async way

    fetch(gitUsersURL)

    // since fetch returns promise wait for it to resolve using .then and convert

    // returned data from api in json object

    .then((res) => res.json())

    // .json method also returns a promise so wait for its results in .then and process

    // returned data

    .then((data) => console.log(data))

    .finally(() => console.log("This is going to execute always at the end"))

}

getGitUsers();

**error example**

fetch doesn’t throw error and invoke .catch() automatically, instead fetch only return response code in case of error (non 200) and then we need to check return code from API and invoke error.

const gitUsersURL = "https://api.github.com/users1"

function getGitUsers(){

    // call api in async way

    fetch(gitUsersURL)

    // since fetch returns promise wait for it to resolve using .then and convert

    // returned data from api in json object

    .then((res) => {

        // we need to explicitly throw error while using fetch by checking return code

        // of api call else return json object

        if(!res.ok){

            throw new Error(res.error);

        }

        return res.json()

    })

    // .json method also returns a promise so wait for its results in .then and process

    // returned data

    .then((data) => console.log(data))

    // handle error in catch

    .catch((err) => console.log(`error occurred invalid api url : ${err}`))

    .finally(() => console.log("This is going to execute always"))

}

getGitUsers();

# Async Await

This makes easy to write promises.

* Async makes a function return promise
* Await makes a function wait for promise

We don’t have to explicitly create a new promise object inside a function and then return it. Instead, we can use this to return a promise.

// async function returns promise

async function sayHello(){

    // creating a new promise because we need mimic long funning 'fetch'

    // operation which returns promise

    const mypromise =  new Promise((resolve, reject) =>{

        // wait for 3 secs to return/resolve the promise

        setTimeout(() => {

            // resolve the promise and return hello

            resolve("Hello There !!!!");

        }, 3000);

    })

    // await is only used inside async function. wait for promise to be resolved/rejected.

    let ret =  await mypromise;

    //return the return value from fulfilled promise

    return ret;

}

// call async function and wait for it to be fulfilled using .then()

sayHello().

then(res => console.log(res));

//=> output is ‘Hello There !!!’ after 3 seconds

Fetch example

async function getUsers(){

    let ret =  await fetch("https://jsonplaceholder.typicode.com/users/11");

    if(ret.ok){

        let users = await ret.json();

        // normal return is treated as promise resolved with api results

        return users;

    }

    else{

        // return explicit promise.reject object to reject the promise

        await Promise.reject(new Error("Error in API !!"));

    }

}

getUsers()

.then(data => console.log(data))

.catch(err => console.log(err));