Js-Essentials3

Async JS

Request

Response

Fetch

Await

What is Async JS?

Async JS is a way to write JavaScript that is `non-blocking`.
`non-blocking` means that the code is not waiting for task completion before continuing.

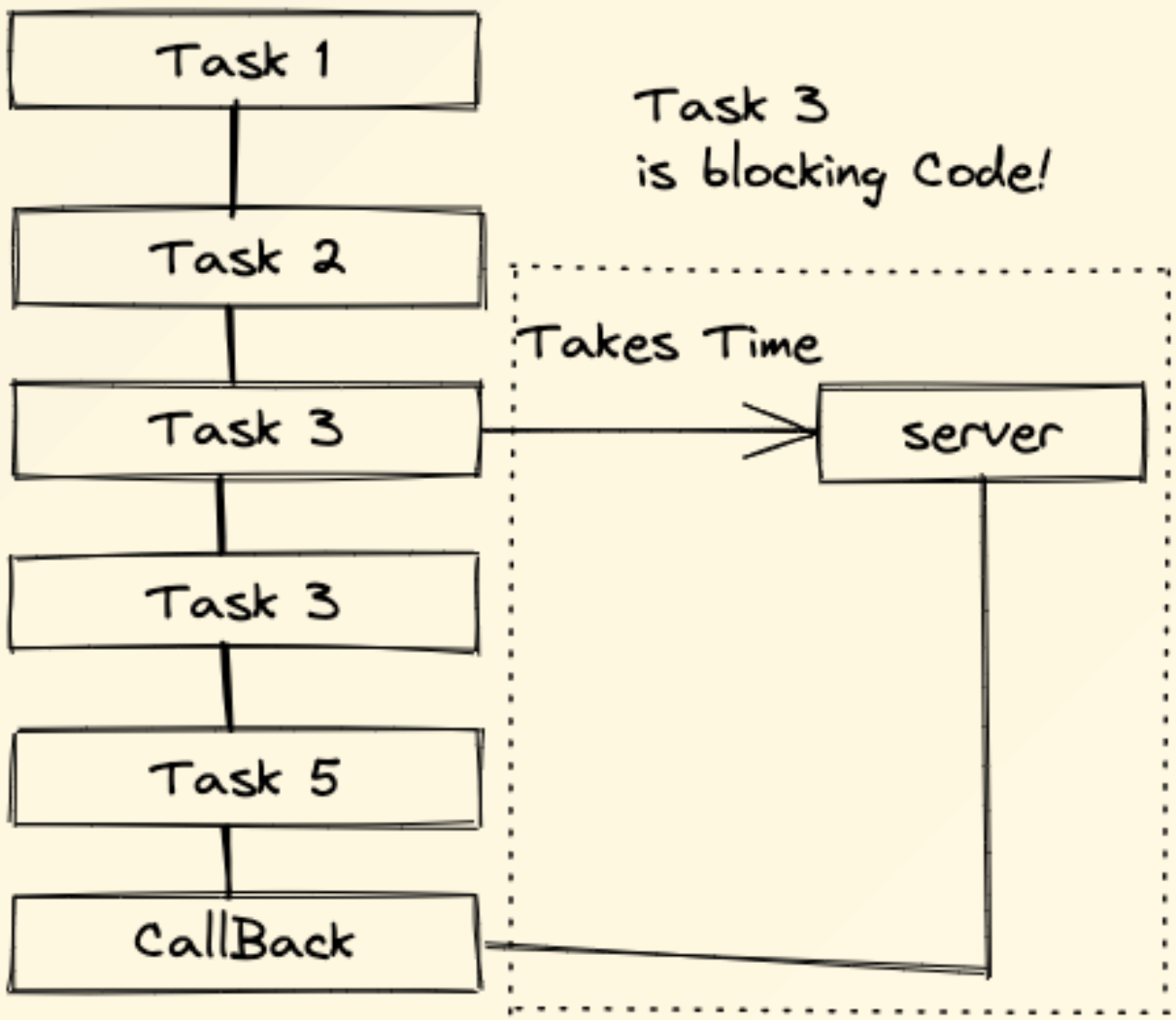
Tasks like fetching data from the server, or waiting for a user to click a button, are not blocking.

> TLDR: Start Some Code finish Later

Threads in JS

- Js is single threaded (only one task can run at a time)
- we can use non blocking code to make our code run in parallel
- imagine set time out is a network request that take 3 seconds to complete
  example of non blocking code:

```
console.log('Task 1');
console.log('Task 2');
setTimeout(() => {
  console.log('callBack');
}, 3000);
console.log('Task 3');
console.log('Task 4');
```



Types of HTTP Requests

- These are actions that the browser can perform on the server
- we want to let say get data from the server; post data to the server; delete data from the server; update data on the server these are the types of HTTP requests
- the request are made to `API endpoints`
- we make a request to gitHub use the API endpoints `https://api.github.com/users/qasimTalkin`
- The data is returned in the form of a JSON object

Chrome Network Tab

- The network tab is where we can see the requests that are made to the server
- In `Header` we can see the type of request that is made
- In `Body` we can see the data that is sent to the server
- In `Response` we can see the data that is returned from the server

```
fetch('https://api.github.com/users/qasimTalkin')
.then(response => response.json())
.then(data => console.log(data))
.catch(err => console.log(err));
```

XMLHttpRequest

- XMLHttpRequest is a way to make a request to the server, this is built into js `let request = new XMLHttpRequest()`
- to set up request `GET` `request.open('GET', 'https://api.github.com/users/qasimTalkin')`
- to send the request `request.send()`
- add event listener to check state of our request
- ready state `0` means request is not initialized, `1` means request has been set up, `2` means request has been sent, `3` means request is in process, `4` means request is complete

```
let request = new XMLHttpRequest();
request.addEventListener('readystatechange', function() {
  if (this.readyState===4){
    console.log( JSON.parse(this.response))
  }
});
request.open('GET', 'https://api.github.com/users/qasimtalkin');
request.send();
```

request.readyState 4 is not enough

- we need to check the status of the request with ready state
- status code `200` means the request was successful
- definition of request status can be found at MDN [MDN Request Status]
- we check for ready status as well this time

```
let request = new XMLHttpRequest();
request.addEventListener('readystatechange', function() {
  if (this.readyState===4 && this.status===200){
    console.log( JSON.parse(this.response))
  }
});
```

creating function to make request

- let `getMyData()` be a function that makes a request to the server
- within getMyData we can set up the request, and send it
- we can pass a callback function to getMyData, this call back will be called when the request is complete

```
let getMyData = function(callback) {
  let request = new XMLHttpRequest();
  request.addEventListener('readystatechange', function() {
    if (request.readyState===4 && request.status===200){
      callback(undefined, request.response)
    } else {
      callback('error', undefined)
    }
  });
  request.open('GET', 'https://api.github.com/users/qasimtalkin');
  request.send();
}
getMyData();
```

response to json

- we can convert the response to json with `JSON.parse(response)`
- the data by default is a string, in order to use the data we need to convert it to json first
  Example

```
let response = JSON.parse(request.response);
// multiple ways to loop through the JSON data
  response.forEach(function(item) {
    console.log(item.name);
  });
  for (item of response) {
    console.log(item.name);
  }
  for (let i = 0; i < response.length; i++) {
    console.log(response[i].name);
  }
```

callBack Hell

- call back hell is when we have nested callbacks, making request to one API and using that data to make another request to another API
- for example we have function that gets all users with an git account and then we have a function that gets all the repositories for each user and then we have a function that gets all the issues for each repository and then we have a function that gets all the comments for each issue
- this is known as call back hell

```
getAllUsers(function(err, users) {
  if (err) {
    console.log(err);
  } else {
    getAllRepos(users[0], function(err, repos) {
      if (err) {
        console.log(err);
      } else {
        getAllIssues(repos[0], function(err, issues) {
          if (err) {
            console.log(err);
          } else {
            getAllComments(issues[0], function(err, comments) {
              if (err) {
                console.log(err);
              } else {
                console.log(comments);
              }
            });
          }
        });
      }
    });
  }
});
```

Promises

- call bak hell can be hard to work with, and we can make it easier with promises
- Promises: a promise is an object that represents the eventual completion or failure of an asynchronous operation
  example

```
// promise with url pram
let promise = (url) => {
  return new Promise(function(resolve, reject) {
    let request = new XMLHttpRequest();
    request.open('GET', url);
    request.onload = function() {
      if (request.status === 200) {
        resolve(request.response);
      } else {
        reject(Error(request.statusText));
      }
    }
    request.onerror = function() {
      reject(Error('Network Error'));
    }
    request.send();
  });
}
```

Fetch API

- fetch is a new way to make a request to the server
- the promise rejected only if there is a network error or if the server returns a status code that is not 200
- we have to make sure the status is
- the response object has information about the response, and in that response object we get json() data in the proto

```
fetch('https://api.github.com/users/qasimtalkin')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(err => console.log(err));
```

Async Await

- async await is a new way to write promises
- it chains promises together in clean and easier way
- we can make the function `async` and then we can use `await` to wait for the promise to resolve
- bundles up all the async code in one function

```
let asyncFunction = async () => {
  let response = await fetch('https://api.github.com/users/qasimtalkin');
  let data = await response.json();
  console.log(data);
}
```

Throwing erros in Async and Await

- we can throw an error in the async function with `throw new Error('error message')`
- we can catch the error in the async function with `try { } catch(err) { }`
- for response object we need to check the status code and throw an error if it is not 200

```
let asyncFunction = async () => {
  let response = await fetch('https://api.github.com/users/qasimtalkin');
  if (response.status !== 200) {
    throw new Error('Request failed with status code ' + response.status);
  }
  let data = await response.json();
  console.log(data);
}
```