# FRONTEND WEB DEVELOPMENT PROGRAM

Lecture 5: JS Async

BeaconFire

# OUTLINE

- Synchronous vs Asynchronous
  Event Loop
- REST, HTTP Requests
  AJAX, XMLHttpRequest
- Promises, Callbacks
  Error Handling
  Fetch vs. XHR
  Async & Await
- Web Storage
- Modules

# SYNCHRONOUS

- **Synchronous code**: executed line-by-line in the order that it is written, and one line of code must finish execution before the next line is executed.

- console.log('First')
  console.log('Second')
  console.log('Third')

- Long-running operations will block code execution.
  - ex: loading a page that requests data from an API

# ASYNCHRONOUS

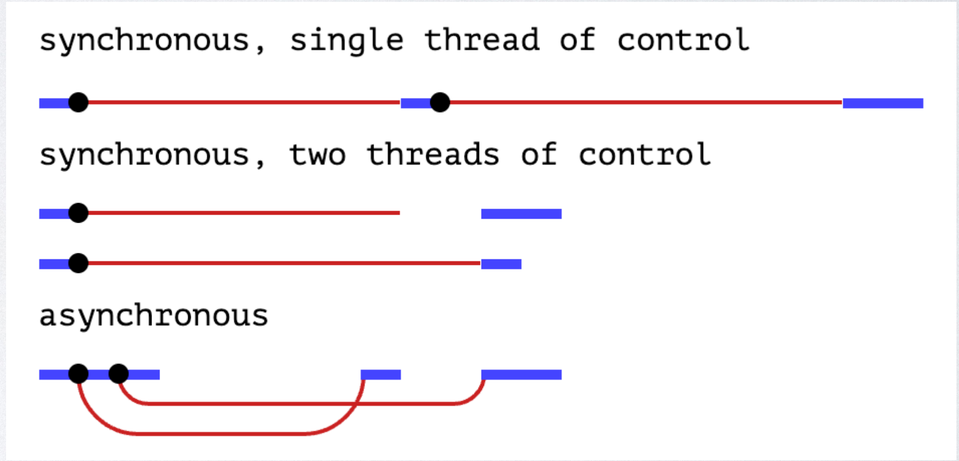
- **Asynchronous code**: a line of code doesn't need to finish execution before the next line is executed.
  - Prevents long-running operations from blocking execution

- JS engine executes line-by-line, but defers execution of the callback function.
  1. Invoke a long-running function and pass in a callback.
  2. It begins the operation and moves on to continue execution.
  3. When the operation completes, execute the callback to finish it off.

```
console.log(`start: ${new Date()}`);
setTimeout(function () {
  console.log(`setTimeout ${new Date()}`);
}, 2000); // 2 second timeout
console.log(`end: ${new Date()}`);
```

# SINGLE-THREADED

- JS is a **single-threaded** programming language.
  - JS engine processes one statement at a time.



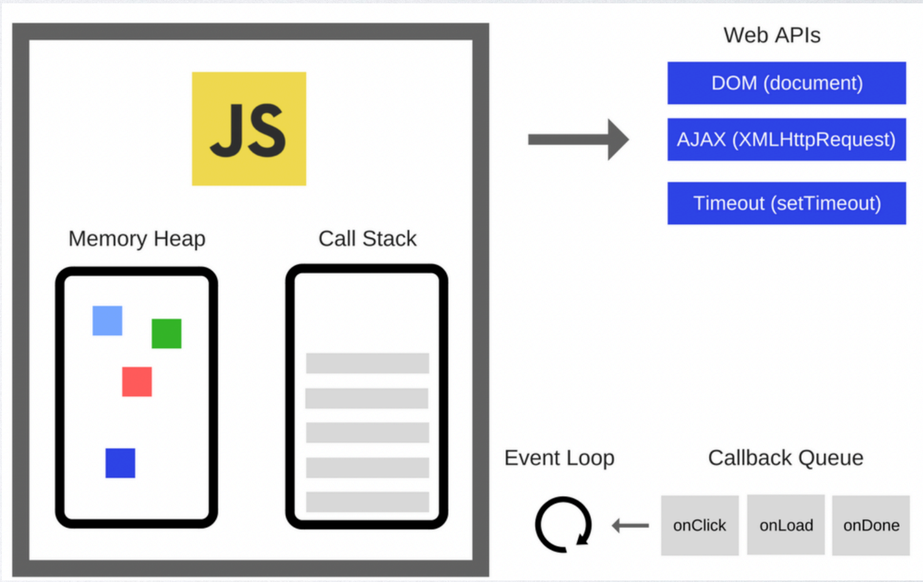
# EVENT LOOP

- **Event loop**: a runtime model that handles execution of synchronous and asynchronous code using these data structures.
  - **Call Stack**
  - **Task Queue** (a.k.a Callback Queue, Message Queue)
    - **Macrotask Queue** (event listener, timeout, interval)
    - **Microtask Queue** (promises)

- (Repeatedly) On empty call stack:
  - Engine processes the next task in the queue
  - Invokes the callback function (creates EC, pushes to stack)

- Order: synchronous > promises > other callbacks
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

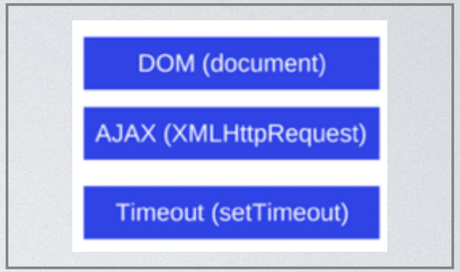
```
while (queue.waitForMessage()) {
  queue.processNextMessage()
}
```



# EVENT LOOP (EXAMPLE)

```
(() => console.log(`start: ${new Date()}`))();
setTimeout(() => console.log(`setTimeout ${new Date()}`));
(() => console.log(`end: ${new Date()}`))();
```

Web APIs




Call stack

Callback Queue

Order of completion

```
(() => console.log(`start: ${new Date()}`))();
(() => console.log(`end: ${new Date()}`))();
setTimeout(() => console.log(`setTimeout ${new Date()}`));
```

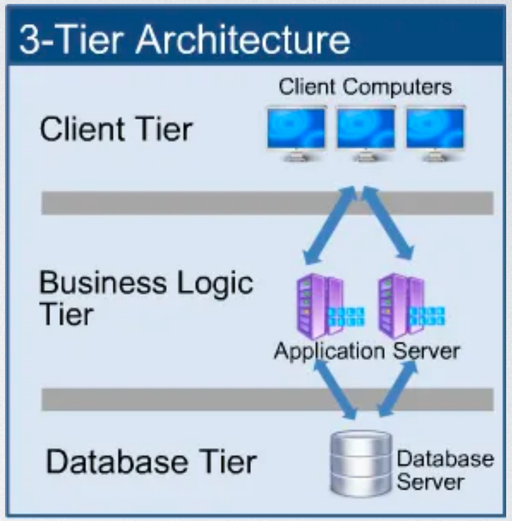
# SERVER REQUEST & RESPONSE

- A webpage often needs access to data. This data might be private user information or hundreds & thousands of items, which should not be stored on the browser due to limited memory, safety concerns, latency…
  - Where is it usually stored? A dedicated data storage (**database**).

- Browsers should not send requests to the database directly.
  - Frontend is responsible for UI/UX, should not have complicated logic for handling data, and is slower than if a dedicated server communicates with the database.

- Solution: Set up a **web server (backend)** that continuously listens for data requests from a **web client (frontend)**. The server handles the request, communicates with the database, and sends the response.

- Frequently: JS sends request => waits (blocks execution) => receives response.
  - Need asynchronous programming.



# REST

- **Representational State Transfer (REST)**: a software architecture for managing internet communications between applications.
  - Uniform interface: servers should transfer information in a standard format
    - **Representation**: data that is sent in a standard format (can differ from the actual structure)
    - Server responses should have enough information that describes how the client should process & manipulate it
  - **Stateless**: server successfully completes each request independently (doesn't store anything related to previous requests)

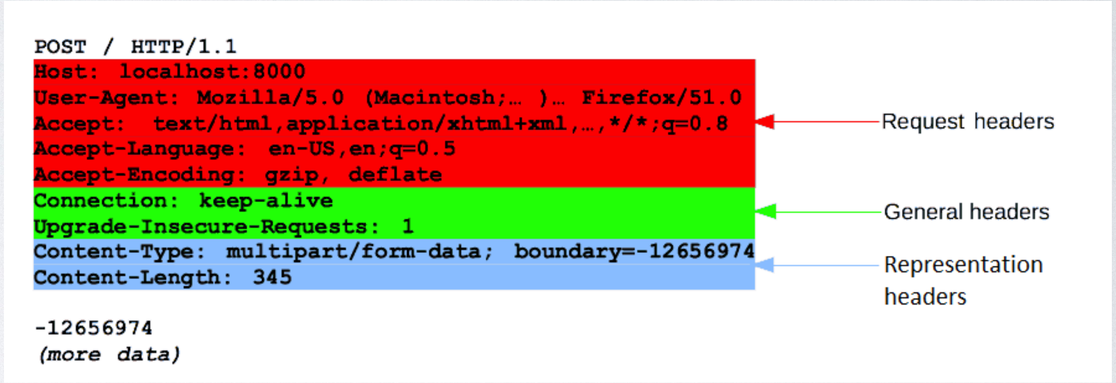
# HTTP

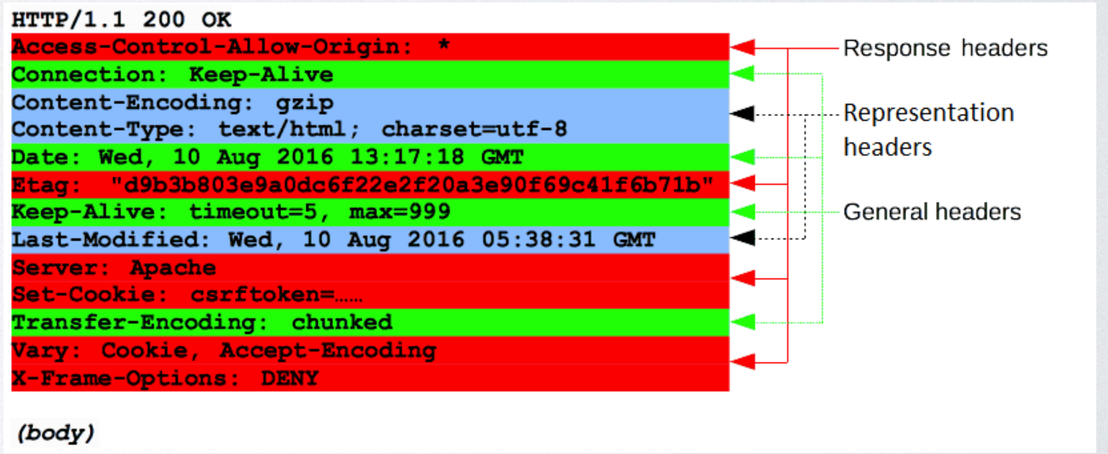
- **Hypertext Transfer Protocol (HTTP)**: a network communication protocol for exchanging information (HTML files, JSON data…) between devices.

- **HTTP Request**: what web clients (browsers) send to a server to retrieve or submit data.

- **HTTP Response**: what web clients (browsers) receive from a server to confirm that their request was received (may contain the requested data).
  - **Status code**: 3-digit codes that indicate whether the request was successfully completed

- **HTTP version, destination url, HTTP method**
  **Headers**: contains core information (browser, data type) in key-value pairs
  **Body (optional)**: the part of the request or response that contains the data




BeaconFire

# HTTP REQUEST METHODS

- **GET**: request data from a specific resource (should not have a body).
  - Typing the url in the browser sends a GET request
- **POST**: send data to a server to create a resource.
  - Can be used to send a data body that identifies resources that you want to retrieve
- **PUT**: send data to a server to replace a resource (for updating).
  - **Idempotent**: sending the same PUT request multiple times always has the same result (no side effects)
- **DELETE**: send data to a server, telling it what resource(s) to delete.
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

# HTTP STATUS CODE

| Code | Meaning | Example |
|---|---|---|
| 100 - 199 | Informational responses | |
| 200 - 299 | Successful responses | 200 OK, 201 Created |
| 300 - 399 | Redirection message | |
| 400 - 499 | Client error responses | 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found |
| 500 - 599 | Server error responses | 500 Internal Server Error, 503 Service Unavailable |

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

# AJAX

- **Asynchronous JavaScript and XML (AJAX)**: using **XMLHttpRequest** objects to asynchronously communicate with servers, exchange data, and update the webpage without reloading.

- **XML (eXtensible markup language)**: used for storing/transporting data.

- We can send/receive information in JSON, XML, HTML, .txt format.

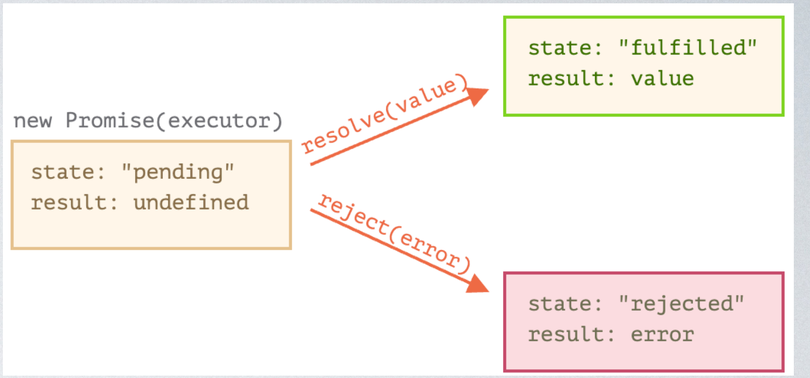
# XML HTTP REQUEST

- **XMLHttpRequest (XHR)**: an object that manages server requests for data. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

- How do we send requests to a server?
  1. Create the request
  2. Configure the request (HTTP method, url, headers)
  3. Send the request
  4. Execute code based on XHR states (success, in progress, error)

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://jsonplaceholder.typicode.com/todoasds/1');
xhr.send();
xhr.onload = function () {
  if (xhr.status != 200)
    console.log(`Error: Status ${xhr.status} ${xhr.statusText}`);
  else {
    const res = JSON.parse(xhr.response);
    const xhrP = document.getElementById('xhrP');
    xhrP.innerText = `Title: ${res.title}`;
  };
};
xhr.onerror = function () {
  console.log("Request failed");
};
xhr.onprogress = function (event) {
  console.log(`Received ${event.loaded} bytes`);
};
```

# PROMISE (ES6)




- **Promise**: an object that represents the eventual completion/failure of an asynchronous operation and its resulting value.
  - 3 states: **Pending**, **Fulfilled**, **Rejected**

- Callback must "**resolve()**" or "**reject()**" with a result or it hangs

- **.then():** invoked on a promise to access its result when it arrives and execute a callback
  - Two optional arguments: success & failure callback
  - Returns a promise

- **Promisify**: turn a function that accepts a callback into a promise.
  - Wrap it in a new promise, and from the callback, resolve() or reject()

```
const promise = new Promise(
  function (resolve, reject) {
    resolve('Success');
    // reject('Error');
});
```

```
promise.then(
  function(result) {
    console.log(`Success: ${result}`)
  },
  function(error) {
    console.log(`Error: ${error}`)
  }
);
```

```
setTimeout(function () {
  promisify.innerText = "Regular setTimeout callback function";
}, 2000);
// ---------
const setTimeoutPromise = new Promise(function (resolve, reject) {
  if (true) {
    setTimeout(function () {
      resolve("Promise setTimeout callback function");
    }, 2000);
  } else reject("Failed");
}).then(function (value) {
  promisify.innerText = value;
});
```

# CONSECUTIVE ASYNC OPERATIONS

- When do we consecutively execute asynchronous operations?
  - Mostly on the backend (invoke many asynchronous operations to deal with data)
- **Callback Hell (ES5)**: nested callback functions whose arguments are results of the outer callback.
  - Difficult to read, maintain, debug
- **Promise chain (ES6)**: consecutively invoking .then() on a promise.
  - Cleaner, more readable, easier debugging & error handling

**Register a new account:**

1. Check if credentials already exist in the database. If not,
2. Encrypt the password.
3. Create the account.
4. Save it to the database.

```
const makeBurger = nextStep => {
  getBeef(function(beef) {
    cookBeef(beef, function(cookedBeef) {
      getBuns(function(buns) {
        putBeefBetweenBuns(buns, beef, function(burger) {
          nextStep(burger);
        });
      });
    });
  });
};
```

```
const makeBurger = () => {
  return getBeef()
    .then(beef => cookBeef(beef))
    .then(cookedBeef => getBuns(beef))
    .then(bunsAndBeef => putBeefBetweenBuns(bunsAndBeef));
};

// Make and serve burger
makeBurger().then(burger => serve(burger));
```

# ERROR HANDLING

- **Error handling**: writing code that processes errors to prevent the program from terminating.

- **try**: wrap error-prone code that'll be executed
  **catch(error)**: execute this code if the try block caused an error
  **finally**: execute this code regardless of errors

```
try {
  console.log("About to execute undefined function.");
  undefinedFunction();
  console.log("Failed to execute undefined function.");
} catch (err) {
  console.log('Caught an error:\n', err);
} finally {
  console.log('This will always execute.');
}
```

- Errors are thrown:
  - implicitly, by the program (reference error, variable not declared)
  - explicitly, by code: **throw**, **reject()**.

- Try-catch handles exceptions (synchronous errors), but not Promise rejections (asynchronous errors).

# ERROR HANDLING (PROMISE)

- **.catch()**: invoked on a promise to catch rejections (asynchronous errors).
  - In promise chain, it catches errors from any promise.
  - You can add the second callback to a .then() for specific error handling, but it continues execution of the chain.

```
new Promise((resolve, reject) => {
    // throw new Error("ERROR");
    reject('ERROR')
}).then((result) => {
  console.log('result:'+result)
}).catch(error=>{
  console.log(`error: ${error}`)
})
```

- Errors that you "**throw**" in asynchronous functions are not caught by try/catch.

# PROMISE.ALL()

- **Promise.all()**: takes an input array of promises and returns a single promise that resolves when they all resolve (all successful).
  - When one promise rejects, Promise.all() rejects.
  - Promise.all() resolves to an array of all the results.

- Use-case: "all-or-nothing" situations where all promises must resolve to continue execution.

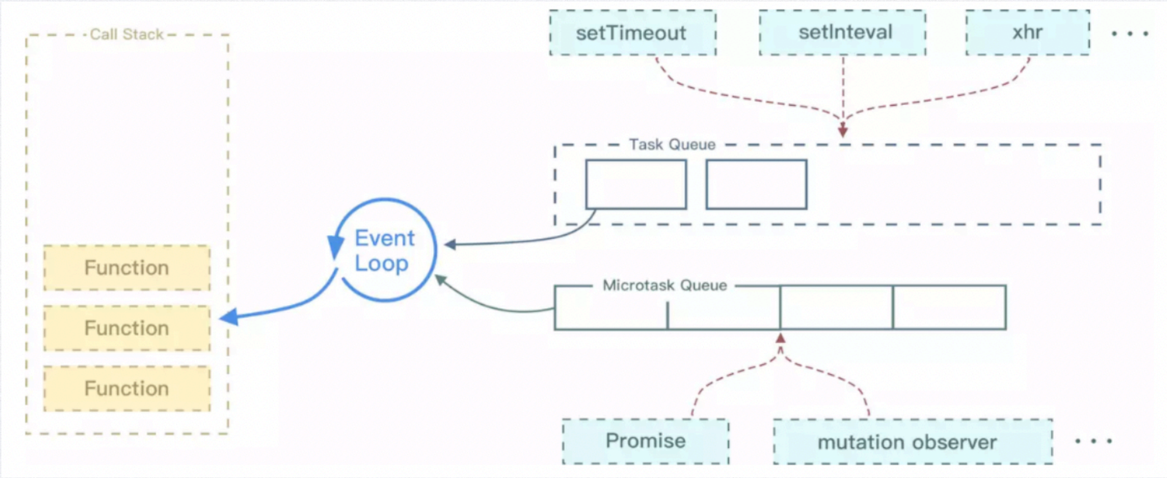
- **Promise.allSettled()**: just like Promise.all() except that it always resolves with an array of all results, giving you access to each promise status & result.

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve(`Promise 1: ${new Date()}`), 2000);
});
const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve(`Promise 2: ${new Date()}`), 1000);
  // setTimeout(() => reject(`Promise 2: ${new Date()}`), 1000);
});
const promise3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve(`Promise 3: ${new Date()}`), 1000);
  // setTimeout(() => reject(`Promise 3: ${new Date()}`), 3000);
});

Promise.all([promise1, promise2, promise3])
  .then(result => console.log("Promise.all() results =", result))
  .catch(result => console.log("Error =", result));
```

# MICROTASK QUEUE

- In the event loop, promises have a **higher priority** than other asynchronous tasks in the Task/Callback/Message Queue, so they execute first.
  - Macrotask Queue (event listener, timeout, interval)
  - **Microtask Queue** (promises)



# FETCH

- **fetch()**: a function used to request resources that returns a promise, which resolves when it has the response.
  - Alternative to XHR and more popular
  - Only rejects on network failure or if anything prevented the request from completing.

- For HTTP response error status codes (ex: 404 Not Found, 500 Internal Server Error), it still resolves, but response**.ok** is false.
  - Manually check response**.ok** and display custom error messages based on response**.status**.

- https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

```
const fetchResult = fetch('https://jsonplaceholder.typicode.com/todos/1');
console.log("fetchResult =", fetchResult); // pending Promise

// Why do we need response.json() ?
fetchResult.then(response => {
  console.log("response =", response); // object: {body, headers, status, statusText}
  console.log("response.json() =", response.json()); // pending Promise
});

// How do we actually get the data?
fetchResult.then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

```
Response {type: 'cors', url: 'https://jsonplaceholder.typicode.com/todos/1sdf
s', redirected: false, status: 404, ok: false, …}
  body: (...)
  bodyUsed: false
  headers: Headers {}
  ok: false
  redirected: false
  status: 404
  statusText: ""
  type: "cors"
  url: "https://jsonplaceholder.typicode.com/todos/1sdfs"
  [[Prototype]]: Response
```

# FETCH VS XHR

```
const btn = document.getElementById('fetchBtn');
btn.addEventListener('click', function () {
  fetch('https://jsonplaceholder.typicode.com/todos/1')
    .then(response => {
      if (response.ok) return response.json();
      else throw new Error(`Error: Status ${response.status}`);
    })
    .then(data => {
      const fetchP = document.getElementById('fetchP');
      fetchP.innerText = `Title: ${data.title}`;
    })
    .catch(error => console.log(error));
});
```

```
const btn = document.getElementById('xhrBtn');
btn.addEventListener('click', function () {
  const xhr = new XMLHttpRequest();
  xhr.open('GET', 'https://jsonplaceholder.typicode.com/todo/1');
  xhr.send();
  xhr.onload = function () {
    if (xhr.status != 200)
      console.log(`Error: Status ${xhr.status} ${xhr.statusText}`);
    else {
      const res = JSON.parse(xhr.response);
      const xhrP = document.getElementById('xhrP');
      xhrP.innerText = `Title: ${res.title}`;
    };
  };
  xhr.onerror = function () {
    console.log("Request failed");
  };
  xhr.onprogress = function (event) {
    console.log(`Received ${event.loaded} bytes`);
  };
});
```

https://stackoverflow.com/questions/35549547/fetch-api-vs-xmlhttprequest

# ASYNC & AWAIT (ES6)

- **async**: a keyword for declaring functions that must return a promise.
  - Return values that aren't promises will be implicitly wrapped with Promise.resolve()
  - Returning a value is like resolve(), "throw" errors is like reject()
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

- **await**: a unary operator that pauses execution of the async function until the given promise completes.
  - Can only be used in async functions or JS modules (imported asynchronously)
  - await on a value that isn't a promise implicitly wraps it with Promise.resolve()
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

- Async/await is a cleaner alternative to promise chains since you don't write callbacks. Wrap everything in async function body in try-catch to handle errors.

```
async function getData(url) {
  try {
    const response = await fetch(url);
    const data = await response.json();
    return data;
  } catch (error) {
    console.log(error);
  }
}
```

# ASYNC/AWAIT VS PROMISE CHAIN

```
// What you'd do with promises
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => response.json())
  .then(data => {
    console.log(data);
    return fetch('https://jsonplaceholder.typicode.com/posts/2');
  })
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

```
// How you'd do it with async-await
(async () => {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
    const data = await response.json();
    console.log(data);

    const response2 = await fetch('https://jsonplaceholder.typicode.com/posts/2');
    const data2 = await response2.json();
    console.log(data2);
  } catch (error) {
    console.log(error);
  }
})();
```

```
// What async-await does under the hood
(() => Promise.resolve()
  .then(() => fetch('https://jsonplaceholder.typicode.com/posts/1'))
  .then(response => response.json())
  .then(data => {
    console.log(data);
    return fetch('https://jsonplaceholder.typicode.com/posts/2');
  })
  .then(response2 => response2.json())
  .then(data2 => {
    console.log(data2);
  })
  .catch(error => console.log(error))
)();
```

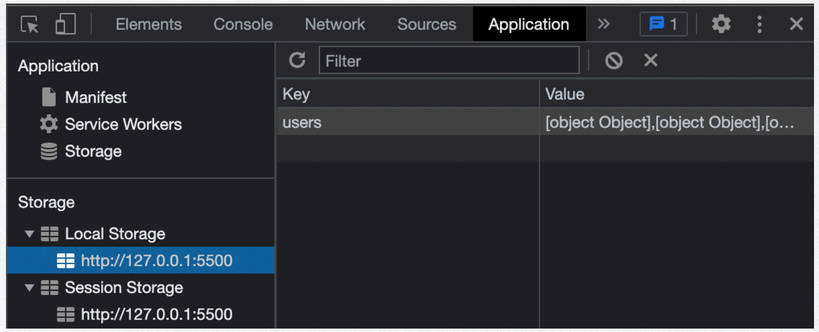
# WEB STORAGE API

- **Web Storage**: simple key-value objects that persist between page load & reload.
  - Keys and values must be strings.

- **sessionStorage**: object that stores data as long as the tab is open (even after page refresh), until the tab is closed.

  **localStorage**: object that stores data (even after browser is closed) until you explicitly clear cache or remove it.

- Storage.**getItem(key)**
  Storage.**setItem(key, value)**
  Storage.**removeItem(key)**
  Storage.**clear()**



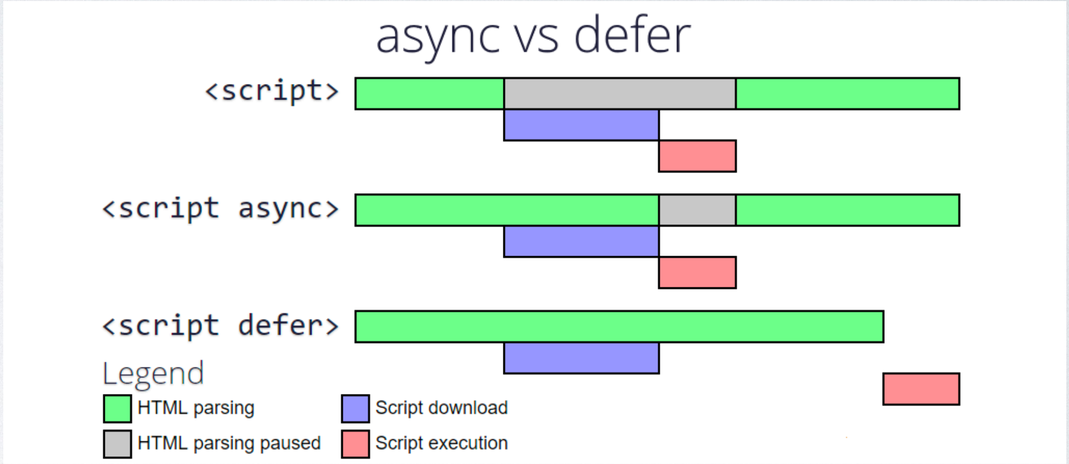

- Use-case: persisting user authentication between page refresh, so they don't need to sign in again

# <SCRIPT>: ASYNC & DEFER

- By default, external scripts (imported with src) are synchronously downloaded and executed, potentially blocking the page render.
  - <link> to stylesheets at the top so they are applied before the page is loaded
  - Scripts are placed at the end of </body> or given "async" or "defer"

- **async**: script is downloaded in parallel and executed right after completion, even if HTML hasn't been fully parsed.
  - May not execute scripts in order
  - Useful if script doesn't manipulate DOM

- **defer**: script is downloaded in parallel, but it will be executed only after HTML is fully parsed.
  - Executes in the order the scripts were imported
  - Useful if script depends on previous script or requires full DOM
  - Same as putting <script> at the end after <body>, but not supported in all browsers




```
<script       src="myscript.js"></script>

<script async src="myscript.js"></script>

<script defer src="myscript.js"></script>
```

# ES MODULES

- **Module**: a unit of code (objects, functions, variables…) with a specific task, several of which are combined to create an application.

- **ES Modules**: one of the JS module systems that are natively supported in browsers as of ES6.
  - Modules are encapsulated in a .js file and are "**export**"ed to other files.
  - In other files, we "**import**" that module to make it available, which happens asynchronously.
    - Must be at the top-level, executes all the code in the imported module.

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

- Why? Maintainability.
  - Large applications => don't want to maintain a file with 1000+ lines of code
  - Each file can have a single purpose.

# ANY QUESTIONS?