Short Answers

1. What are the disadvantages of synchronous code?

Synchronous code is executed line-by-line in the order it is written, meaning one line of code must complete execution before the next line starts. If we run a line of code that, for example, requests data from an API, all subsequent tasks will have to wait until the request finishes. On the client-side, synchronous code can freeze the UI, causing unresponsiveness, for example, when a user attempts repeated mouse clicks. Synchronous execution can also lead to increased latency and longer total execution time when fetching multiple API endpoints sequentially, as each request must wait for the previous one to complete. Overall, it's just slower when all tasks have to wait for one task to finish.

2. What is asynchronous code in JavaScript?

Asynchronous code in JavaScript is when a line of code does not need to finish execution before the next line is executed. This prevents long-running operators from blocking execution and making the program unresponsive. So when it comes to tasks like `setTimeout`s, `addEventListener`s, or promises, these can run asynchronously while synchronous code runs line by line.

3. How does JavaScript achieve asynchronous code?

JavaScript is a single-threaded programming language; its engine processes one statement at a time. But JavaScript is able to achieve asynchronous behaviour through the event loop, making the code *feel* like it's doing multiple tasks at once.

4. What does the event loop do? What data structures does it use?

The event loop is a runtime model that handles execution of synchronous and asynchronous code in JavaScript using the call stack and the task queue. The task queue is further divided into two categories: the macrotask queue and the microtask queue. The macrotask queue runs things like `setTimeout`s, `setInterval`s, `addEventListener`s, while the microtask queue runs things like promises and has a higher priority than the macrotask queue.

5. What is the callback queue?

The callback queue is part of JavaScript's event loop mechanism and holds callback functions that are waiting to be executed after the currently executing code (synchronous tasks) and any higher-priority tasks, such as microtasks, that have been processed. The callback queue is essentially the same thing as the task queue, as these two terms can be used interachangelably.

6. What is an HTTP request and HTTP response?

HTTP stands for Hypertext Transfer Protocol. It is a network communication protocol for exchanging information between devices, such as HTML files, JSON data, and other resources over the web. HTTP involves both HTTP requests and HTTP responses. An HTTP request is what web clients (the browser) sends to the server to retrieve or submit data. An HTTP response is what web clients (the browser) receive from a server to confirm that their request was received (and may contain the requested data). Other things

that are usually involved incldue a server status code that is a three-digit number ranging from 100 to 599, a destination URL, and an HTTP method.

7. How many HTTP methods are there? Explain each one. What is the difference between GET and POST? What about POST and PUT?

The five HTTP methods are `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`, which correspond to the CRUD operations: create, read, update, and delete. `GET` "gets" or reads data from the server, doesn't modify or create anything, and usually uses the `200` server status code. `POST` "posts" or creates new data on the server, usually comes with a body, and uses the `201` server status code. Both `PUT` and `PATCH` send data to the server telling it to update an existing resource with new data, often comes with a body, and usually uses the `200` server status code. But `PUT` is for replacing the entire resource with the new resource, whereas `PATCH` only updates part of the resource. And finally, `DELETE` sends data to a server telling it what resource to delete, usually uses the `200` server status code, and does not come with a body.

8. Could you explain the different classes of HTTP status codes? What are some common status codes?

Server status codes fall under five categories: 100 to 199 for information responses, 200 to 299 for successful responses, 300 to 399 for redirectional messages, 400 to 499 for client-error responses, and 500 to 599 for server error responses. The more common status codes are `200` for OK, the request was successful and the server is returning the requested data; `201` for created, the request was successful and a the resource was created; `301` for moved permanently, the resource has been permanently moved to a new URL; `400` for bad request, the server cannot process the request because something wrong was entered; `401` for unauthorized, the request requires the user to be authenticated by providing valid credentials such as username and password; `403` for forbidden, the user is authenticated and the server understood the request, but the user's role does not have access to this data; `404` for not found, the server could not find the requested resource; `500` for internal server error, the server encountered an error while processing the request; and `503` for service unavailable, the server is temporarily unavailable (undergoing maintenance or is down).

9. What is AJAX?

AJAX stands for Asynchronous JavaScript and XML. AJAX uses XHR objects to asynchronously communicate with servers, exchange data (usually in JSON format today), and update the webpage without reloading. XML stands for eXtensive Markup Language and is used for storing and transporting data. We can send and receive information in `JSON`, `XML`, `HTML`, and `.txt` formats.

10. What is XHR?

XHR stands for XMLHttpRequest and is an object that manages server requests for data and is the technology behind AJAX. How XHR works:

1. Create an XMLHttpRequest object to initiate a request.
2. Configure the request by specifying the request type (`GET`, `POST`, `PUT`, or `DELETE`), the URL to fetch data from, and the headers.
3. Send the request to the server.
4. Execute the code based on XHR states (success, in progress, or error).

11. What is a Promise?

A promise (introduced in ES6) is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value.

12. How many states does a Promise have? What are they?

A promise has three states: "pending", which is the initial state of the promise, and from "pending", it can only go to either "fulfilled" (the promise successfully completed) or "rejected" (the promise failed).

13. What is callback hell?

Callback hell refers to nested callback functions whose arguments are the results of the outer callback. They are difficult to read, maintain, and debug. The following code is an exampel of callback hell:

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

14. What is the advantage of Promises over callbacks?

The advantage that Promises have over callbacks is when we need to execute consecutive asynchronous operations. Promises allow for promise chaining, where the `.then()` method can be consecutively invoked on a promise. This makes promises cleaner, more readable, and easier to debug and handle errors. The following code takes the above example and converts it into promise chaining:

```
const makeBurger = () => {
    return getBeef()  // assuming this returns a promise
        .then(beef => cookBeef(beef))  // pass beef to cookBeef
        .then(cookedBeef => getBuns())  // get buns after cooking beef
        .then(buns => putBeefBetweenBuns(buns, cookedBeef))  // pass both
buns and cooked beef
        .then(burger => burger);  // return final burger
};
makeBurger().then(burger => server(burger));  // pass burger to the server
```

15. Explain Promise.all() vs Promise.allSettled().

`Promise.all()` takes an array of promises and returns a single promise. This returned promise resolves when all the input promises resolve successfully. If any of the promises are rejected, `Promise.all()` will be rejected immediately with the reason of the first promise that got rejected. If all promises are successful, `Promise.all()` resolves to an array of the results from each promise, in the same order as the input promises.

`Promise.allSettled()` is similar to `Promise.all()`, but with a key difference: it always resolves with an array of all results, regardless of whether the promises were successful or rejected. Each result in the array is an object that describes the status of the promise ("fulfilled" or "rejected") and its corresponding value or reason. This means that even if one or more promises are rejected while others are successful, you will still receive information about each promise's outcome, and the successful promises will show up as "fulfilled".

16. What is the Microtask Queue?

The microtask queue is part of the event loop and holds tasks such as promises (specifically, the resolution of promises). It has a higher priority than the macrotask queue (which holds tasks like `setTimeout` or `setInterval`), meaning that tasks in the microtask queue are executed before tasks in the macrotask queue. When a promise is resolved, its `.then()` or `.catch()` callback is added to the microtask queue, and these callbacks are executed after the currently executing synchronous code, but before the next macrotask.

17. What is the difference between making server requests via fetch and XHR?

`fetch()` is a function used to request resources that returns a promise, which resolves when it receives a response. `fetch()` uses promises under the hood (it's built on promises). It is an alternative to XHR and is more popular because it is cleaner, simpler, and easier to handle asynchronous operations. `fetch()` only rejects on a network failure or if something prevents the request from completing. The following are examples of a `GET` request using fetch in a project of mine, which retrieves a list of games, and a `POST` request that creates a review for a game:

```
// initiates a GET request to fetch a list of games from the API
fetch("https://vapor-al92.onrender.com/api/games/all", {
    method: "GET"
})
    .then(response => response.json())  // convert response to JSON format
    .then(data => console.log(data))  // capture the data
    .catch(error => console.error("Fetch Error: ", error))  // catches and
logs any errors


// initiates a POST request to create a review for a game
fetch("https://vapor-al92.onrender.com/api/games/1/review/post", {
    method: "POST",
    credentials: "include",
    headers: {
        "Content-Type": "application/json",
        "XSRF-TOKEN": "insert-actual-token-here"
```

```
    },
    body: JSON.stringify({
        thumbs_up: true,
        thumnbs_down: false,
        description: "This is a test review for game ID 1."
    })
})
    .then(response => {
        if (response.ok) return response.json()
        else throw new Error(`Error: ${response.status}
${response.statusText}`)
    })
    .then(data => console.log(data))
    .catch(error => console.error("Fetch Error: ", error))
```

18. What is async & await? How do we use them?

`async` is the keyword used to declare functions that always return a promise. If a function returns a value that is not a promise, it is automatically wrapped in `Promise.resolve()`. Returning a value is equivalent to resolving a promise, and throwing an error is like rejecting a promise.

`await` is an operator that pauses the execution of an `async` function until the promise passed to it resolves or rejects. `await` can only be used inside `async` functions or in JavaScript modules that are imported asynchronously. If you `await` a value that isn't a promise, it will automatically be wrapped in `Promise.resolve()`.

`async/await` provides a cleaner alternative to promise chains since you don't need to write callbacks. Instead, you wrap everything in an `async` function and handle errors using try-catch blocks as in the following code:

```
(async () => {
  try {
    const response1 = await
fetch('https://jsonplaceholder.typicode.com/posts/1');
    const data = await response1.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);
  }
})();
```