CONCISE BOOKLET WEB DEV

Web Development

Table of Contents

Fundamentals & Basics

HTML & CSS

1. What does HTML stand for?

   HTML stands for **HyperText Markup Language**.

2. What is the purpose of the `<head>` tag in HTML?

   The `<head>` tag contains metadata, like the document title, character set, styles, scripts, and other information not directly displayed on the webpage. learn more

3. How do you create a hyperlink in HTML?

   Use the `<a>` tag with the `href` attribute. learn more

   `<a href="https://www.example.com">Link Text</a>`.

4. What is the role of the `<alt>` attribute in an `<img>` tag?

   The `alt` attribute provides alternative text for an image if it cannot be displayed. It's also important for accessibility. learn more

5. How do you create a list with bullets in HTML?

   Use the `<ul>` tag for an unordered (bulleted) list and `<li>` tags for list items. learn more

   ```
   <ul>
   <li>Item 1</li>
   <li>Item 2</li>
   </ul>
   ```

6. What is the difference between `<div>` and `<span>`?

   `<div>` is a block-level element used for grouping larger sections, while `<span>` is an inline element used to style smaller text or parts of a sentence. learn more

7. How do you add comments in HTML?

   Use `<!-- comment here -->` for comments in HTML.

8. What is the purpose of the `<DOCTYPE html>` declaration?

   It tells the browser to render the document in standards mode, treating it as HTML5. learn more

9. How can you embed a video in HTML?

   Use the `<video>` tag with the `src` attribute or nested `<source>` tags. learn more

   ```
   <video controls>
   <source src="video.mp4" type="video/mp4">
   </video>
   ```

10. What tag is used for creating a form in HTML?

    The `<form>` tag is used to create a form. It can include form elements like `<input>`, `<button>`, `<select>`, etc. learn more

11. What does CSS stand for?

    CSS stands for **Cascading Style Sheets**.

12. How do you add CSS to an HTML document?

    There are three ways: learn more

    - **Inline CSS:** using the `style` attribute in HTML tags.
    - **Internal CSS:** within a `<style>` tag in the HTML `<head>`.
    - **External CSS:** linking an external CSS file with `<link rel="stylesheet" href="style.css">`.

13. What is the purpose of the `display` property in CSS?

    The `display` property defines how an element should behave visually, such as `block`, `inline`, `flex`, or `none`. learn more

14. How can you center a block element horizontally in CSS?

    By setting `margin: 0 auto;` and specifying a width.

    ```
    .center {
    width: 50%;
    margin: 0 auto;
    }
    ```

15. What is the difference between `id` and `class` selectors in CSS?

    An `id` is unique to a single element and is selected with `#`, while a `class` can be used on multiple elements and is selected with `.`. learn more

16. What is a CSS pseudo-class? Give an example

    A pseudo-class is a keyword added to selectors that specifies a special state of the selected elements.

    Example:

    ```
    a:hover
    ```

    applies styles when a link is hovered. learn more

17. How do you make a responsive design in CSS?

    Using media queries, flexbox, and/or CSS grid to adjust layout for different screen sizes. learn more

18. What does the `z-index` property do?

    It controls the stack order of elements. Higher values place elements above those with lower values. learn more

19. How can you apply transparency to a background color in CSS?

    Use `rgba` for color values with an alpha channel. Example: `background-color: rgba(0, 0, 0, 0.5);`. learn more

20. What is the difference between `padding` and `margin`?

    `Padding` is the space inside an element, between the content and the border. `Margin` is the space outside the border, separating the element from others.

Tutorial : HTML, CSS

Javascript

1. What is JavaScript, and what is it primarily used for?

   JavaScript is a scripting language that allows you to implement complex features on web pages, like dynamic content, interactivity, animations, and more.

2. How do you declare a variable in JavaScript?

   You can declare a variable using `var`, `let`, or `const`. Example: `let x = 10;`. learn more

3. What's the difference between `let`, `const`, and `var`?

   - `let` allows you to declare variables that are block-scoped.
   - `const` is similar to `let` but is used for variables that should not be reassigned.
   - `var` is function-scoped, meaning it is accessible within a function, but has more flexible scoping rules than `let` and `const`. learn more

4. What is a function, and how do you define one in JavaScript?

   A function is a reusable block of code that performs a specific task.

   ```
   function greet() {
   console.log("Hello!");
   }
   ```

   learn more

5. What is an arrow function?

   An arrow function is a shorthand way to write functions using `=>`.

   ```
   const greet = () => console.log("Hello!");
   ```

   learn more

6. What is the purpose of `this` in JavaScript?

   `this` refers to the context in which a function is called. It can refer to the global object, an object instance, or a specific element based on the call context. learn more

7. What are JavaScript data types?

   JavaScript data types include **String**, **Number**, **Boolean**, **Object**, **Array**, **Function**, **Null**, **Undefined**, **Symbol** (ES6), and **BigInt** (ES11). learn more

8. What's the difference between `==` and `===` in JavaScript?

`==` checks for equality with type coercion, whereas `===` checks for strict equality without type coercion. learn more

9. How do you create an array in JavaScript?

   Arrays are created using square brackets `[]`. Example: `let fruits = ["apple", "banana", "cherry"];`.

   learn more

10. What are template literals, and how do you use them?

    Template literals are strings enclosed in backticks (

    ```
    `
    ```

) and allow embedded expressions with

    ```
    ${}
    ```

    .

    Example:

    ```
    `Hello, ${name}!`
    ```

    . learn more

11. What are `null` and `undefined` in JavaScript?

    `null` is an assignment value that represents "no value," while `undefined` indicates a variable has been declared but not assigned a value.

    learn more

12. What is a callback function?

    A callback function is a function passed as an argument to another function to be executed after some operation has completed.learn more

13. Explain the concept of closures in JavaScript.

    A closure is a function that retains access to its lexical scope, even when executed outside of its original scope. This allows inner functions to access variables from an outer function.learn more

14. How can you check if a variable is an array in JavaScript?

    Use `Array.isArray(variable)` to check if a variable is an array.

15. What are ES6 modules, and how do you use them?

    ES6 modules allow you to export and import code between JavaScript files.

    ```
    // file1.js
    export const greet = () => console.log("Hello!");
    ```

    ```
    // file2.js
    import { greet } from './file1.js';
    greet();
    ```

13. How do you handle asynchronous code in JavaScript?

    Asynchronous code can be handled using callbacks, Promises, or `async/await` syntax.

learn more

14. What is the purpose of `JSON.stringify()` and `JSON.parse()`?
    - `JSON.stringify()` converts a JavaScript object into a JSON string.
    - `JSON.parse()` parses a JSON string back into a JavaScript object.
15. How do you add a new element to the end of an array?

    Use the `push()` method.

    ```
    let numbers = [1, 2, 3];
    numbers.push(4); // [1, 2, 3, 4]
    ```

    learn more
16. What are `map()`, `filter()`, and `reduce()` used for in JavaScript?
    - `map()` creates a new array by applying a function to each element.
    - `filter()` creates a new array with elements that pass a test.
    - `reduce()` reduces the array to a single value by executing a reducer function.

    learn more
17. What is event delegation in JavaScript?

    Event delegation is a pattern where a single event listener is added to a parent element, managing events on its child elements through the event's target property.

    learn more

Tutorial : https://www.youtube.com/watch?v=hdI2bqOjy3c

React js

1. What is React.js?

   React.js is a JavaScript library developed by Facebook for building user interfaces. It's component-based and allows developers to build reusable UI components.
2. What are components in React?

   Components are the building blocks of a React application. Each component is an independent, reusable piece of UI that can have its own state and logic.

   learn more
3. What is the difference between functional and class components?
   - **Class Components** use ES6 classes, require `render()` method to return JSX, and have lifecycle methods.
   - **Functional Components** are simpler, just functions that return JSX, and can use hooks (like `useState` and `useEffect`) for state and lifecycle features.

   learn more
4. What is JSX?

   JSX (JavaScript XML) is a syntax extension that allows you to write HTML-like code inside JavaScript files. It's used in React to describe the UI. learn more

5. What are props in React?

   Props (short for "properties") are inputs to components that allow data to be passed from one component to another, typically from parent to child.

   learn more

6. What is state in React?

   State is a built-in object that allows components to manage and update data that affects the component's rendering. State is mutable and can be updated with `setState` or `useState` in functional components.

   learn more

7. How does the `useState` hook work?

   `useState` is a hook in React that allows functional components to use state. It returns an array with two elements: the current state and a function to update it.

   ```
   const [count, setCount] = useState(0);
   ```

   learn more

8. What is the virtual DOM?

   The virtual DOM is a lightweight copy of the real DOM. React uses the virtual DOM to track changes, and only updates the actual DOM where changes occurred, making updates more efficient.

   learn more

9. What is `useEffect` in React?

   `useEffect` is a hook that performs side effects in functional components, such as data fetching, subscriptions, and DOM manipulations. It runs after each render by default.

   learn more

10. How do you create a React app?

    The easiest way is to use `create-react-app` by running `npx create-react-app my-app`. This sets up a new React project with all necessary configurations.

    learn more

11. What is the purpose of keys in React?

    Keys help React identify which elements have changed, are added, or are removed. Keys should be unique, stable identifiers for each element in a list to optimize rendering.

    learn more

12. What is the `props.children` in React?

    `props.children` is a special prop that allows you to pass JSX or other components as the children of a component.

    learn more

13. What is the purpose of `useContext`?

    `useContext` is a hook that allows components to access values from React's Context API, providing a way to share values (like themes or user data) without passing props down manually at every level. learn more

14. What is the difference between controlled and uncontrolled components?

    - **Controlled Components** have their state managed by React using `useState` or `setState`.

- **Uncontrolled Components** rely on the DOM for managing state, and `ref` is used to access their values.

learn more

15. What are React fragments, and why use them?

    React Fragments (`<></>`) allow you to group multiple elements without adding extra nodes to the DOM. They're helpful for returning multiple elements in a component without a wrapping div.

    learn more

16. What is `lifting state up` in React?

    Lifting state up is a pattern where the state is moved up to the closest common ancestor of components that need access to it. This allows multiple components to share and sync state.

    learn more

17. What are higher-order components (HOCs)?

    HOCs are functions that take a component and return a new component with enhanced functionality. They are used to reuse component logic.

    learn more

18. How does React handle events?

    React events are similar to HTML events but are named in camelCase and prevent default behavior by calling `e.preventDefault()`. Event handlers are passed as functions.

    learn more

19. What is the `React Router` used for?

    React Router is a library for adding routing to a React application, allowing navigation between pages or views without refreshing the page.

    learn more

20. What is Redux, and how does it work with React?

    Redux is a state management library that helps manage and centralize application state. With React, it allows components to access shared state globally through actions and reducers.

Tutorial : https://www.youtube.com/watch?v=LDB4uaJ87e0

Django

1. What is Django?

   Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It's known for being secure, scalable, and fully-featured.

2. What are models in Django?

   Models in Django define the structure of the database tables and the attributes of the data. Each model is a Python class that subclasses `django.db.models.Model` .learn more

3. What is the purpose of `django-admin` ?

   `django-admin` is a command-line utility for Django that allows you to perform administrative tasks like creating projects, starting apps, and running development servers.

learn more

4. How do you create a new Django project?

   Use `django-admin startproject projectname` to create a new project with necessary configurations and files.

5. What is an app in Django?

   An app in Django is a web application that does something specific, like handling blog posts or user authentication. A project can contain multiple apps.

   learn more

6. How do you create a new app in Django?

   Use `python manage.py startapp appname` to create a new app within a Django project.

7. What is the role of `urls.py` in Django?

   `urls.py` is used for URL routing. It maps URLs to views so that different parts of an application can be accessed via specific URLs.learn more

8. How do you define a model in Django?

   Models are defined by creating a class that subclasses `models.Model`. Each attribute of the model represents a field in the database.

   ```
   from django.db import models
   class Blog(models.Model):
   title = models.CharField(max_length=200)
   content = models.TextField()
   ```

   learn more

9. How do you run Django migrations?

   Run `python manage.py makemigrations` to create migrations and `python manage.py migrate` to apply them to the database.learn more

10. What is the purpose of `settings.py`?

    `settings.py` contains the configuration for a Django project, including database settings, static file locations, installed apps, and middleware configurations.

    learn more

11. What are views in Django?

    Views handle the logic for processing user requests and returning responses. They can be function-based or class-based.

    learn more

12. What is a Django template?

    Templates are HTML files that allow dynamic content rendering by using Django Template Language (DTL). They define the structure and layout of the front end of a Django app.

    learn more

13. What is ORM in Django?

    ORM (Object-Relational Mapping) allows Django to interact with the database using Python classes and methods rather than writing SQL queries. It makes data manipulation easier and database-agnostic.

    learn more

14. How do you pass context data to a template in Django?

    You can pass data to a template by creating a dictionary and passing it through the view function's `render()` method.

    ```
    def my_view(request):
    context = {'name': 'John'}
    return render(request, 'template.html', context)
    ```

15. What are Django static files?

    Static files are files like CSS, JavaScript, and images. Django has a specific way to manage and serve static files using the `STATIC_URL` and `STATICFILES_DIRS` settings.

    learn more

16. What is the Django admin interface?

    Django's built-in admin interface allows developers to manage application data and content easily. It's highly customizable and can be enabled with minimal setup.

    learn more

17. How do you register a model in Django admin?

    To register a model in the Django admin, import it in `admin.py` and use `admin.site.register(ModelName)` to make it accessible.

    ```
    from .models import Blog
    admin.site.register(Blog)
    ```

18. What is middleware in Django?

    Middleware is a way to process requests and responses globally. It is a layer that lies between the request and the view, and can handle various tasks like security, session management, and authentication.

    learn more

19. What is the purpose of the `manage.py` file in Django?

    `manage.py` is a command-line utility that allows developers to interact with the Django project by running tasks such as starting the server, applying migrations, and creating apps.

    learn more

20. What are querysets in Django?

    Querysets are a way to retrieve data from the database in Django. They allow filtering, ordering, and chaining operations to access model instances from the database in an optimized way.

    learn more

Tutorial : https://www.youtube.com/watch?v=e1IyzVyrLSU

SQL

1. What is SQL?

   SQL (Structured Query Language) is a standard language for managing and manipulating relational databases. It includes commands for data retrieval, data manipulation, and data definition

2. What are the main types of SQL commands?

- **DDL (Data Definition Language):** `CREATE`, `ALTER`, `DROP`
- **DML (Data Manipulation Language):** `INSERT`, `UPDATE`, `DELETE`
- **DQL (Data Query Language):** `SELECT`
- **DCL (Data Control Language):** `GRANT`, `REVOKE`

3. What is a primary key in SQL?

   A primary key uniquely identifies each row in a table. It must contain unique values and cannot contain `NULL`.learn more

4. How do you use the `SELECT` statement in SQL?

   The `SELECT` statement retrieves data from a database.

   ```
   SELECT column_name FROM table_name;
   ```

   learn more

5. What does `JOIN` do in SQL?

   `JOIN` combines rows from two or more tables based on a related column. Common types include `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `FULL JOIN`.

   learn more

6. What is the difference between `WHERE` and `HAVING`?

   `WHERE` filters rows before grouping, while `HAVING` filters groups created by `GROUP BY`.

   learn more

7. How do you create a new table in SQL?

   Use the `CREATE TABLE` statement.

   ```
   CREATE TABLE table_name (
   column1 datatype,
   column2 datatype,
   ...
   );
   ```

8. What does the `GROUP BY` clause do?

   `GROUP BY` groups rows with the same values in specified columns, often used with aggregate functions like `SUM()` and `COUNT()`

   learn more

9. What is a foreign key?

   A foreign key is a field in one table that links to the primary key of another table, establishing a relationship between the two tables.

   learn more

10. What is a `NULL` value in SQL?

    `NULL` represents missing or unknown data. It's not the same as zero or an empty string.

11. How do you update data in SQL?

    Use the `UPDATE` statement.

    ```
    UPDATE table_name
    SET column1 = value1, column2 = value2
    WHERE condition;
    ```

learn more

12. What is indexing in SQL?

    Indexing improves data retrieval speed by creating a sorted structure for specific columns. However, it can slow down `INSERT` and `UPDATE` operations.

    learn more

13. What is normalization in SQL?

    Normalization organizes a database to reduce redundancy and dependency. Levels include 1NF, 2NF, 3NF, and BCNF.

    learn more

14. What is a transaction in SQL?

    A transaction is a sequence of database operations executed as a single unit, ensuring ACID (Atomicity, Consistency, Isolation, Durability) properties.

    learn more

15. How do you delete data in SQL?

    Use the `DELETE` statement to remove specific rows.

    ```
    DELETE FROM table_name WHERE condition;
    ```

    learn more

16. What is a stored procedure?

    A stored procedure is a set of SQL statements saved in the database that can be executed to perform a specific task.

    learn more

17. What is the difference between `UNION` and `UNION ALL`?

    `UNION` combines results from multiple queries and removes duplicates, while `UNION ALL` includes all results, including duplicates.

    learn more

18. What is a view in SQL?

    A view is a virtual table based on the result of a SQL query. It allows for simplified data access and security.

    learn more

19. How does the `LIKE` operator work?

    `LIKE` is used for pattern matching with wildcards `%` (any number of characters) and `_` (a single character).

    ```
    SELECT * FROM table_name WHERE column_name LIKE 'pattern';
    ```

    learn more

20. What is a subquery in SQL?

    A subquery is a query nested within another SQL query, used to filter or compute intermediate results.

    learn more

Tutorial : https://www.youtube.com/watch?v=nWeW3sCmD2k

MongoDB

1. What is MongoDB?

   MongoDB is a NoSQL document database that stores data in JSON-like, flexible documents. It's known for its scalability and performance.

2. What is a collection in MongoDB?

   A collection is a group of MongoDB documents, similar to a table in SQL databases, but it can store documents with varying structures

   learn more

3. What is a document in MongoDB?

   A document is a JSON-like object in MongoDB that stores data as key-value pairs, equivalent to a row in an SQL table

   learn more

4. How do you insert a document into a MongoDB collection?

   Use the `insertOne()` or `insertMany()` method.

   ```
   db.collectionName.insertOne({ name: "John", age: 30 });
   ```

5. How do you query data in MongoDB?

   Use the `find()` method to retrieve documents.

   ```
   db.collectionName.find({ name: "John" });
   ```

6. What is indexing in MongoDB?

   Indexing in MongoDB improves search performance by creating an ordered structure for specific fields, but it may increase storage and impact write performance.

   learn more

7. How do you update documents in MongoDB?

   Use `updateOne()` or `updateMany()` to modify documents.

   ```
   db.collectionName.updateOne({ name: "John" }, { $set: { age: 31 } });
   ```

   learn more

8. What are operators in MongoDB?

   MongoDB operators are used to filter data (`$gt` , `$lt`), update documents (`$set` , `$unset`), and perform various operations.

   learn more

9. How does the `$set` operator work in MongoDB?

   `$set` updates specific fields in a document without affecting other fields.

   ```
   db.collectionName.updateOne({ name: "John" }, { $set: { age: 35 } });
   ```

10. What is the `_id` field in MongoDB?

    `_id` is a unique identifier for each document in MongoDB, automatically generated if not provided.

11. How do you delete a document in MongoDB?

    Use `deleteOne()` or `deleteMany()` to remove documents.

    ```
    db.collectionName.deleteOne({ name: "John" });
    ```

learn more

12. What is aggregation in MongoDB?

    Aggregation is a framework for data processing and analysis, using stages like `$match`, `$group`, and `$project` to transform data.

    learn more

13. What is the `$match` stage in MongoDB aggregation?

    `$match` filters documents based on specified conditions, similar to `WHERE` in SQL.

14. What is a replica set in MongoDB?

    A replica set is a group of MongoDB servers that maintain copies of the same data, providing redundancy and high availability.

    learn more

15. How does sharding work in MongoDB?

    Sharding distributes data across multiple servers to support horizontal scaling, dividing large datasets into smaller parts.

16. What is the purpose of `findOne()` in MongoDB?

    `findOne()` retrieves a single document from a collection that matches the specified filter.

    learn more

17. What is a schema in MongoDB?

    MongoDB is schema-less, meaning it does not enforce a fixed schema, allowing documents in the same collection to have different structures.

    learn more

18. What is a pipeline in MongoDB aggregation?

    A pipeline is a sequence of stages that process documents. Each stage performs an operation on the documents and passes them to the next stage.

    learn more

19. How do you create a text index in MongoDB?

    Use the `createIndex()` method with the `{ text: "text" }` option to enable text search on a field.

20. What are MongoDB transactions?

    Transactions in MongoDB allow multiple operations to be executed in an all-or-nothing manner, ensuring data integrity, especially in distributed environments.

Tutorial : https://www.youtube.com/watch?v=QPFlGswpyJY

Web Security

1. What is web security?

Web security involves practices and techniques to protect websites, web applications, and online services from cyber threats, ensuring data confidentiality, integrity, and availability.

2. What is the CIA triad in cybersecurity?

   The CIA triad stands for Confidentiality, Integrity, and Availability. It's a fundamental model in cybersecurity to guide policies and practices:

   - **Confidentiality:** Ensuring that sensitive information is accessible only to authorized users.
   - **Integrity:** Maintaining the accuracy and trustworthiness of data.
   - **Availability:** Ensuring systems and data are accessible when needed.

   learn more

3. What is Cross-Site Scripting (XSS)?

   XSS is a vulnerability where an attacker injects malicious scripts into a trusted website, potentially allowing them to steal data or hijack user sessions.

   learn more

4. How can XSS be prevented?

   Preventing XSS involves:

   - Validating and escaping user input
   - Using Content Security Policy (CSP) headers
   - Avoiding the insertion of untrusted data into the HTML structure.

5. What is Cross-Site Request Forgery (CSRF)?

   CSRF is an attack that tricks an authenticated user into performing unwanted actions on a web application, potentially compromising their account.

   learn more

6. How can CSRF attacks be prevented?

   CSRF prevention techniques include:

   - Using CSRF tokens for form submissions
   - Requiring user re-authentication for critical actions
   - Implementing SameSite cookies.

7. What is SQL Injection?

   SQL Injection (SQLi) is a vulnerability that allows an attacker to inject malicious SQL code into a query, potentially accessing or manipulating the database.

   learn more

8. How can SQL Injection be mitigated?

   Mitigation techniques for SQL Injection include:

   - Using parameterized queries or prepared statements
   - Avoiding dynamic SQL queries

- Validating and sanitizing user input.

9. What is a Denial-of-Service (DoS) attack?

   A DoS attack aims to overwhelm a server, network, or service with excessive traffic, causing it to become unavailable to legitimate users.

   learn more

10. What is HTTPS, and why is it important?

    HTTPS (Hypertext Transfer Protocol Secure) encrypts data transmitted between a user's browser and the server, protecting it from eavesdropping and man-in-the-middle attacks.

    learn more

11. What is Clickjacking?

    Clickjacking is a technique where an attacker tricks users into clicking on hidden elements on a page, leading to actions that the user did not intend to perform, such as approving permissions.

    learn more

12. How can Clickjacking be prevented?

    Clickjacking prevention methods include:

    - Using `X-Frame-Options` HTTP headers to control framing
    - Implementing a Content Security Policy (CSP) with frame restrictions.

13. What is an Authentication Bypass?

    An authentication bypass allows an attacker to gain unauthorized access to a system without the need for valid credentials, typically due to a flaw in the authentication process.

    learn more

14. What is a Man-in-the-Middle (MITM) attack?

    In a MITM attack, an attacker intercepts and potentially alters communication between two parties without their knowledge, potentially stealing or modifying data.

    learn more

15. How can Man-in-the-Middle attacks be prevented?

    MITM attack prevention includes:

    - Using HTTPS/TLS to encrypt data in transit
    - Employing strong encryption for wireless networks (WPA3)
    - Avoiding unsecured networks for sensitive transactions.

16. What is a firewall, and how does it work?

    A firewall is a network security device that monitors and filters incoming and outgoing traffic based on security rules, helping prevent unauthorized access to or from a private network.

    learn more

17. What is an API rate limit, and why is it important?

API rate limiting restricts the number of requests a client can make within a specified time frame, protecting APIs from abuse, DoS attacks, and excessive resource consumption.

learn more

18. What is Two-Factor Authentication (2FA)?

    2FA is an additional security layer requiring two forms of identification (such as a password and a one-time code) to verify a user's identity.

    learn more

19. What is Open Redirect, and how can it be prevented?

    Open Redirect occurs when an application allows users to redirect to an untrusted site, potentially leading to phishing attacks. Prevent it by validating and restricting redirect URLs.

    learn more

20. What is the purpose of a Content Security Policy (CSP)?

    CSP is a security measure that restricts the sources from which content (scripts, styles, etc.) can be loaded on a web page, helping to prevent XSS and data injection attacks.

    learn more

RESTful API's and GraphQL

1. What is a RESTful API?

   A RESTful API is an application programming interface (API) that follows the principles of Representational State Transfer (REST) architecture. It enables communication between a client and server by using HTTP methods like GET, POST, PUT, and DELETE.

2. What are the main HTTP methods used in RESTful APIs, and what do they do?

   The main HTTP methods are:

   - **GET**: Retrieves data from the server.
   - **POST**: Sends new data to the server.
   - **PUT**: Updates existing data on the server.
   - **DELETE**: Removes data from the server.

   learn more

3. What are endpoints in RESTful APIs?

   Endpoints are specific URLs that represent different resources within a RESTful API. Each endpoint corresponds to a particular function, such as fetching or updating data, and is accessed via an HTTP request.

   learn more

4. What is an API response status code? Give some examples.

   An API response status code is a code returned by the server to indicate the result of a request. Examples include:

   - **200 OK**: The request was successful.

- **201 Created**: A new resource was created.
- **400 Bad Request**: The request was invalid.
- **404 Not Found**: The resource was not found.
- **500 Internal Server Error**: A server error occurred.

learn more

5. What are request headers, and why are they important in RESTful APIs?

   Request headers contain metadata sent with an HTTP request. They provide essential information, like content type (`Content-Type: application/json`), authorization tokens, and accepted response formats, which help servers process requests correctly.

   learn more

6. How is authentication commonly handled in RESTful APIs?

   Common authentication methods in RESTful APIs include:

   - **Token-based authentication**: Using a token (e.g., JWT) passed in headers.
   - **OAuth**: Delegated access via third-party tokens.
   - **Basic authentication**: Base64 encoded username and password.

   learn more

7. What is statelessness in RESTful APIs?

   Statelessness means each API request from a client must contain all the information the server needs to fulfill it. No client state is stored on the server between requests, making each request independent.

   learn more

8. What is JSON, and why is it commonly used in RESTful APIs?

   JSON (JavaScript Object Notation) is a lightweight, text-based data format used for data exchange. It's popular in RESTful APIs because it's easy to read, write, and parse across various programming languages.

   learn more

9. What are RESTful resources, and how are they typically represented?

   RESTful resources are entities exposed via the API, like users, products, or orders. They are typically represented as JSON objects and accessed through URL endpoints (e.g., `/api/users`).

10. What is versioning in RESTful APIs, and why is it important?

    Versioning allows developers to introduce new features or make changes without disrupting existing API users. Common approaches include URL versioning (e.g., `/api/v1/resource`) and header-based versioning.

    learn more

11. What is GraphQL, and how does it differ from REST?

    GraphQL is a query language for APIs that enables clients to request only the specific data they need. Unlike REST, which has fixed endpoints, GraphQL allows clients to define the structure of the response, reducing over-fetching and under-fetching of data.

12. What is a GraphQL schema?

A GraphQL schema defines the types of data and the relationships between them in an API. It also specifies the queries and mutations (operations) that clients can perform, acting as the blueprint of the API.

learn more

13. What are queries in GraphQL?

    Queries in GraphQL are requests for data. They allow clients to specify the structure of the response, selecting only the fields they need from the available types in the schema.

    learn more

14. What are mutations in GraphQL?

    Mutations are operations in GraphQL used to modify data (create, update, or delete) on the server. Each mutation defines the data it accepts and the data it returns after execution.

    learn more

15. How does GraphQL handle errors?

    In GraphQL, errors are returned as part of the response, in an `errors` field separate from the `data` field. This enables the client to receive partial data even if some fields encounter errors.

    learn more

16. What are resolvers in GraphQL?

    Resolvers are functions that handle fetching the actual data for each field in a query or mutation. They map schema fields to backend data sources and can fetch data from databases, other APIs, or even perform calculations.

    learn more

17. What are the main benefits of using GraphQL over REST?

    Benefits include:

    - **Precise data fetching**: Clients request only the data they need.
    - **Single endpoint**: All data requests are handled through a single endpoint.
    - **Strongly typed schema**: Ensures clear structure and documentation

    learn more

18. What is a GraphQL fragment, and how is it used?

    A fragment is a reusable piece of a query that defines a set of fields. Fragments allow for consistent data requests across different parts of an application, simplifying query maintenance.

    learn more

19. How is pagination handled in GraphQL?

    GraphQL supports pagination with arguments like `first`, `last`, `before`, and `after`, typically paired with `edges` and `nodes` structures. Cursor-based pagination is commonly used for performance and flexibility.

    learn more

20. What is a subscription in GraphQL?

    A subscription is a real-time operation that enables clients to receive updates whenever specific data changes. Unlike queries and mutations, subscriptions use WebSocket connections instead of HTTP.

learn more

Intermediate level questions

HTML & CSS

1. How can you create a responsive image gallery using CSS Grid? Provide an example code snippet.

   You can create a responsive image gallery by defining a grid layout that adapts to different screen sizes using media queries. Here's an example:

   ```
   <div class="gallery">
   <img src="image1.jpg" alt="Image 1">
   <img src="image2.jpg" alt="Image 2">
   <img src="image3.jpg" alt="Image 3">
   <!-- More images -->
   </div>
   ```

   ```
   .gallery {
   display: grid;
   gap: 10px;
   grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
   }
   .gallery img {
   width: 100%;
   height: auto;
   border-radius: 8px;
   }
   ```

learn more

2. Explain the CSS `position` property values (`static`, `relative`, `absolute`, `fixed`, and `sticky`) with examples.

   Here's a summary of the CSS `position` property values:

   - `static`: Default position; elements are placed in the normal document flow.
   - `relative`: Positioned relative to its normal position.
   - `absolute`: Positioned relative to the nearest positioned ancestor (not in the normal flow).
   - `fixed`: Positioned relative to the viewport, remaining fixed during scroll.
   - `sticky`: Behaves like `relative` until it reaches a scroll position, where it then becomes fixed.

   learn more

3. What are CSS pseudo-classes and pseudo-elements?

   Pseudo-classes are used to style elements based on their state. Example: `:hover`

   Pseudo-elements style a part of an element, such as `::before`.

   learn more

4. How can CSS variables improve code maintainability

   CSS variables, defined with `--variable-name`, centralize commonly-used values, making updates easier.

   ```
   :root {
   --primary-color: #3498db;
   --padding: 10px;
   }
   ```

```
button {
background-color: var(--primary-color);
padding: var(--padding);
}
```

5. What is Flexbox, and how does it differ from CSS Grid? When would you choose one over the other?

   Flexbox is a one-dimensional layout system, useful for aligning items in a row or column, while CSS Grid is a two-dimensional system, better for complex layouts. Flexbox is ideal for simpler, linear arrangements, and CSS Grid is preferred for full-page layouts with rows and columns.learn more

6. Explain how the `box-sizing` property works and why it's commonly set to `border-box`.

   `box-sizing: border-box` includes padding and border in the element's specified width, preventing layout issues when adding padding or borders

   learn more

7. How can you create a pure CSS tooltip that appears on hover?

   Tooltips can be created by showing a `::after` element on hover:

```
<div class="tooltip">Hover over me
<span class="tooltip-text">Tooltip text</span>
</div>
```

```
.tooltip {
position: relative;
display: inline-block;
cursor: pointer;
}
.tooltip .tooltip-text {
visibility: hidden;
position: absolute;
bottom: 100%;
left: 50%;
transform: translateX(-50%);
background-color: #333;
color: #fff;
padding: 5px;
border-radius: 3px;
white-space: nowrap;
}
.tooltip:hover .tooltip-text {
visibility: visible;
}
```

8. How does the CSS `calc()` function work?

   `calc()` allows mathematical calculations within CSS, helpful for dynamic sizing:

```
.box {
width: calc(100% - 40px);
padding: calc(10px + 1%);
}
```

   learn more

9. Describe how to create a responsive navbar using only CSS.

   A simple responsive navbar can use a combination of Flexbox and media queries

   learn more

10. Explain CSS animations and keyframes

CSS animations are used to create effects by changing styles over time. Keyframes define the stages of an animation.

learn more

Javascript

1. How do closures work in JavaScript? Provide an example to illustrate how a closure captures variables from its lexical scope.

   A closure occurs when an inner function has access to the variables of an outer function, even after the outer function has finished executing. Here's an example:

   ```
   function createCounter() {
   let count = 0;
   return function() {
   count++;
   return count;
   };
   }
   ```

   ```
   const counter = createCounter();
   console.log(counter()); // Output: 1
   console.log(counter()); // Output: 2
   ```

   Here, `counter` is a closure that "remembers" the `count` variable, allowing it to maintain state across multiple calls.

2. What is the difference between `call()`, `apply()`, and `bind()`?

   All three methods are used to change the `this` context of a function, but they differ in their syntax and usage.

   - `call()` invokes the function immediately with arguments passed individually.
   - `apply()` invokes it with arguments passed as an array.
   - `bind()` returns a new function with the `this` context set, and can take arguments that are passed when the function is called.

   learn more

3. How does the `this` keyword work in JavaScript?

   `this` refers to the context in which a function is called:

   - **In global scope**, `this` refers to the global object (`window` in browsers).
   - **In an object method**, `this` refers to the object.
   - **In a class instance**, `this` refers to the instance.
   - **With `bind`, `call`, or `apply`**, `this` can be explicitly set

4. Explain how JavaScript's asynchronous nature works, especially with Promises. How do `async` and `await` make handling asynchronous code easier?

   JavaScript uses an event loop to handle asynchronous operations. Promises represent a value that may be available now, later, or never.

   `async` / `await` syntax makes asynchronous code look synchronous, improving readability by avoiding `.then()` chains.

   learn more

5. How do you handle errors in JavaScript, especially in asynchronous functions?

Errors in synchronous code are handled with `try...catch`. In asynchronous code, `try...catch` can be combined with `async/await`.

learn more

6. Describe JavaScript's `spread` and `rest` operators

   The `spread` operator expands elements of an iterable, while the `rest` operator collects all remaining elements into an array.

   ```
   const arr = [1, 2, 3];
   const newArr = [...arr, 4, 5]; // Spread: [1, 2, 3, 4, 5]
   ```

   ```
   function sum(...nums) { // Rest
   return nums.reduce((acc, num) => acc + num, 0);
   }
   ```

   ```
   console.log(sum(1, 2, 3, 4)); // Output: 10
   ```

   learn more

7. What is `debouncing` in JavaScript?

   Debouncing is a technique to limit how often a function is called. It waits until the last call in a series of calls to execute the function, helpful in optimizing event handlers.

   learn more

8. How does JavaScript handle prototypes, and what's the difference between `__proto__` and `prototype`?

   In JavaScript, every object has a prototype, providing inheritance. `prototype` is an object that's used to build `__proto__` when an object is created.

   - `__proto__`: The actual object representing the prototype of another object.
   - `prototype`: A property on constructor functions that's used to define methods for instances created by that constructor.

   learn more

React JS

1. What are React hooks, and how do `useState` and `useEffect` work

   React hooks allow functional components to have state and other React features without needing class components.

   - `useState`: Manages state within a functional component.
   - `useEffect`: Runs side effects in functional components (e.g., data fetching, subscriptions).

   learn more

2. Explain the purpose of `useReducer`

   `useReducer` provides an alternative to `useState` for more complex state logic, useful for managing complex state like form inputs.

   learn more

3. Describe memoization in React and how `React.memo` can optimize functional components.

   `React.memo` is a higher-order component that prevents a component from re-rendering if its props haven't changed. This improves performance by memoizing the component's output.

   ```
   import React from 'react';
   ```

```
const Child = React.memo(({ value }) => {
console.log('Child rendered');
return <div>Value: {value}</div>;
});
```

```
function Parent() {
const [count, setCount] = useState(0);
return (
<div>
<Child value={count} />
<button onClick={() => setCount(count + 1)}>Increment</button>
</div>
);
}
```

learn more

4. How does React's `useRef` hook work

   `useRef` creates a mutable reference object that persists across re-renders. It's commonly used to directly access DOM elements.

   learn more

5. Explain what React portals

   React portals allow rendering a component's output outside its parent component's DOM hierarchy, useful for modals and tooltips.

   learn more

6. How would you handle side effects and cleanup in a functional component

   Side effects (like data fetching) are managed using `useEffect`, which can also handle cleanup to avoid memory leaks.

   learn more

7. How does lazy loading work in React, and why is it useful?

   Lazy loading in React delays the loading of a component until it's needed, which can improve performance.

   learn more

8. Explain Prop Drilling in React

   Prop drilling occurs when data is passed through multiple components to reach a deeply nested component. The Context API can help by providing a way to pass data through the component tree without manually passing props at every level

   learn more

Django

1. How do you create a custom Django model manager, and why would you use one?

   A custom model manager in Django allows you to encapsulate common queries as methods, providing a cleaner, more reusable way to access filtered or complex queries.

   ```
   from django.db import models
   ```

   ```
   class PublishedManager(models.Manager):
   def get_queryset(self):
   return super().get_queryset().filter(status='published')
   ```

   ```
   class Post(models.Model):
   title = models.CharField(max_length=100)
   body = models.TextField()
   status = models.CharField(max_length=10)
   ```

```
published = PublishedManager()
```

https://www.geeksforgeeks.org/built-in-custom-model-managers-in-django/

2. Explain Django's signal system

   Signals in Django allow you to react to certain actions in the lifecycle of a model (like saving or deleting). The `post_save` signal can trigger functions immediately after a model instance is saved.

   https://medium.com/jungletronics/how-django-signals-work-81dc30d0dad5

3. What is the purpose of Django middleware

   Middleware in Django processes requests and responses, allowing you to add custom behaviors at various points in the request-response cycle.

4. How do you create a custom template filter in Django

   Custom template filters allow you to create reusable logic for templates. For example, you might want a filter that truncates long text to a certain number of characters.

   ```
   from django import template
   register = template.Library()
   @register.filter
   def truncate_chars(value, num):
   return value[:num] + '...' if len(value) > num else value
   ```

   https://www.geeksforgeeks.org/custom-template-filters-in-django/

5. What is the purpose of Django's `GenericForeignKey`

   `GenericForeignKey` allows a model to reference multiple other models without a fixed foreign key relationship. This is useful for models that can be related to different types of content.

   https://dipbazz.medium.com/how-and-when-to-use-genericforeignkey-in-django-ad88202be0f

6. How do you use Django's `select_related` and `prefetch_related` to optimize database queries?

   `select_related` and `prefetch_related` are used for optimizing database access by reducing the number of queries in one-to-many and many-to-many relationships.

   ```
   posts = Post.objects.all()
   for post in posts:
   print(
   post.author.name )  # Causes N+1 queries
   ```

   # `With select_related (for ForeignKey fields)`

   ```
   posts = Post.objects.select_related('author').all()
   ```

   learn more

7. Explain how class-based views (CBVs) work in Django

   CBVs provide a way to structure views with less code and more flexibility. For example, `DetailView` automatically handles retrieving a single object and passing it to a template.

   https://www.geeksforgeeks.org/class-based-generic-views-django-create-retrieve-update-delete/

8. How does Django's `get_object_or_404` function work

   `get_object_or_404` retrieves an object if it exists or raises a `404` error if not, simplifying error handling and improving code readability.

   ```
   from django.shortcuts import get_object_or_404
   from .models import Post
   ```

```
def post_detail(request, post_id):
post = get_object_or_404(Post, id=post_id)
return render(request, 'post_detail.html', {'post': post})
```

This function avoids needing to manually check if `post` exists and handle `None`, reducing boilerplate code.

SQL/MongoDB

1. How do you perform a complex query in MongoDB to filter documents based on conditions in embedded documents?

   You can use the `$elemMatch` operator to filter based on conditions in embedded arrays.

   ```
   db.orders.find({
   items: {
   $elemMatch: { qty: { $gte: 5 }, price: { $lt: 100 } }
   }
   })
   ```

   This query retrieves all orders with at least one item where `qty` is 5 or more and `price` is less than 100.

2. How can you update multiple documents in MongoDB to add a new field if it doesn't exist

   You can use the `$set` operator along with the `upsert: false` option in an `updateMany` operation.

   ```
   db.products.updateMany(
   { price: { $exists: true } },
   { $set: { discount: 0 } },
   { upsert: false }
   )
   ```

   This query adds a `discount` field with a value of `0` to all products that have a `price` field.

3. Explain how MongoDB's `$lookup` operator works and write a basic aggregation with a lookup.

   `$lookup` performs a join-like operation to pull data from another collection.

   ```
   db.orders.aggregate([
   {
   $lookup: {
   from: "customers",
   localField: "customer_id",
   foreignField: "_id",
   as: "customer_info"
   }
   }
   ])
   ```

   This pipeline joins `orders` and `customers` collections, adding `customer_info` as a field in each order document.

   learn more

4. How do you create an index on multiple fields in MongoDB, and why would you do this?

   Multi-field indexes, or compound indexes, optimize queries filtering on multiple fields.

   ```
   db.products.createIndex({ category: 1, price: -1 })
   ```

   This index optimizes queries filtering by `category` and `price`. The `1` denotes ascending order, and `-1` denotes descending.

5. **: How do you use MongoDB's `$group` operator in an aggregation to get the sum of a field?**

   Use `$group` to create aggregations based on specific fields, with `$sum` to total a field.

   ```
   db.sales.aggregate([
   { $group: { _id: "$product_id", totalQuantity: { $sum: "$quantity" } } }
   ])
   ```

This groups sales by `product_id` and calculates `totalQuantity` for each product.

learn more

6. How can you ensure uniqueness on a field in MongoDB?

   Use a unique index on the field to enforce uniqueness.

   ```
   db.users.createIndex({ email: 1 }, { unique: true })
   ```

   This ensures that no two documents in the `users` collection will have the same email.

7. Explain how to use MongoDB transactions and why they are useful

   Transactions allow multiple operations to be executed atomically, ensuring data integrity across collections.

   ```
   const session = client.startSession();
   session.startTransaction();
   try {
   db.orders.insertOne({ order_id: 1, items: [...] }, { session });
   db.inventory.updateOne({ product_id: 1 }, { $inc: { stock: -1 } }, { session });
   session.commitTransaction();
   } catch (error) {
   session.abortTransaction();
   } finally {
   session.endSession();
   }
   ```

   This ensures that if one operation fails, the other is also rolled back.

8. How do you delete documents in MongoDB based on a field's value range?

   Use `deleteMany` with a range condition.

   ```
   db.logs.deleteMany({ createdAt: { $lt: new Date('2024-01-01') } })
   ```

   This deletes all log entries created before January 1, 2024.

9. How can you update a specific element in an array in a MongoDB document?

   Use the positional `$` operator to target array elements.

   ```
   db.courses.updateOne(
   { _id: 1, "
   students.name
   ": "Alice" },
   { $set: { "students.$.grade": "A" } }
   )
   ```

   This updates `Alice`'s grade to "A" within the `students` array of course `1`.

10. **How can you get a count of documents that match certain criteria in MongoDB?A:** Use the `countDocuments` function for an efficient count.

    ```
    db.orders.countDocuments({ status: "delivered" })
    ```

    This returns the number of orders with a `delivered` status.

11. **How do you retrieve the top 5 employees with the highest salaries in SQL?**

    Use `ORDER BY` with `LIMIT` to retrieve the top records.

    ```
    SELECT name, salary
    FROM employees
    ORDER BY salary DESC
    LIMIT 5;
    ```

    This selects the names and salaries of the top 5 highest-paid employees.

12. Explain what a subquery is and give an example of using it in SQL

    A subquery is a query nested within another SQL query, often used for filtering or aggregating.

```
SELECT name
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

13. How do you perform a join operation to get data from multiple tables? Provide an example with an `INNER JOIN`.

    Use `JOIN` clauses to retrieve related data from multiple tables.

    ```
    SELECT orders.order_id,  customers.name
    FROM orders
    INNER JOIN customers ON orders.customer_id = customers.customer_id;
    ```

    This fetches order IDs and customer names by joining `orders` and `customers` tables on `customer_id`.

14. What is a common table expression (CTE) in SQL

    A CTE allows you to define a temporary result set within a query.

    learn more

Web Security

1. What is Cross-Site Scripting (XSS) and how can you prevent it in a web application?

   XSS occurs when an attacker injects malicious scripts into webpages viewed by other users. Preventing XSS involves validating and escaping user input, using secure libraries, and enabling Content Security Policy (CSP).

   ```
   // Example of escaping in JavaScript
   function escapeHTML(str) {
   return str.replace(/&/g, "&")
   .replace(/</g, "<")
   .replace(/>/g, ">")
   .replace(/"/g, """)
   .replace(/'/g, "'");
   }
   ```

   This code escapes special characters, mitigating the risk of XSS by preventing scripts from executing in user input fields

2. What is SQL Injection, and how can you protect against it in a database-driven application?

   SQL Injection allows an attacker to execute arbitrary SQL commands by injecting SQL code into user input. Use prepared statements and parameterized queries to prevent SQL injection.

   ```
   cursor.execute("SELECT * FROM users WHERE username = %s", (username,))
   ```

   By using parameterized queries, the input is treated as data rather than SQL commands, preventing malicious injection.

3. Explain the concept of Cross-Site Request Forgery (CSRF) and how it can be mitigated.

   CSRF tricks a user into performing actions on a site they are authenticated on without their consent. To prevent CSRF, include a CSRF token in requests that validate the user's intent.

   ```
   <!-- Example form with CSRF token in Django -->
   <form method="post" action="/submit-data/">
   {% csrf_token %}
   <input type="text" name="data">
   <input type="submit">
   </form>
   ```

   CSRF tokens are unique to each session and ensure the request comes from the authenticated user, protecting against CSRF attacks.

4. What is Content Security Policy (CSP) and how does it protect web applications?

   CSP is an HTTP header that restricts resources the browser can load for a page, mitigating XSS and data injection attacks by defining which sources are trustworthy.

```
Content-Security-Policy: default-src 'self'; img-src 'self' https://images.com ; script-src 'self'
```

This CSP policy only allows scripts and images from the same origin and from trusted domains, reducing the risk of XSS.

5. How would you secure sensitive data like passwords and tokens stored in a web application?

   Store sensitive data using strong encryption algorithms (e.g., AES for tokens, bcrypt for passwords), avoid storing plain text, and limit access.

   ```
   import bcrypt
   hashed = bcrypt.hashpw(password.encode(), bcrypt.gensalt())
   ```

   By hashing passwords with bcrypt, they are stored securely and can't be easily reverted to plain text, even if the database is compromised.

6. What are Same-Origin Policy (SOP) and CORS, and why are they important for web security?

   SOP prevents a site from making requests to different origins, protecting against data theft. CORS (Cross-Origin Resource Sharing) selectively loosens SOP by allowing trusted domains.

   ```
   // Example of setting CORS headers in an Express.js application
   app.use((req, res, next) => {
   res.header("Access-Control-Allow-Origin", "
    https://trustedwebsite.com ");
   next();
   });
   ```

   This configuration allows only `https://trustedwebsite.com` to make requests to the server, reducing the risk of cross-origin attacks.

7. Describe the principles of HTTPS and TLS, and explain how HTTPS improves web security.

   HTTPS uses TLS to encrypt data transferred between a client and server, protecting against eavesdropping and man-in-the-middle attacks. TLS also authenticates the server's identity with certificates.

   ```
   openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout mykey.key -out mycert.crt
   ```

   HTTPS encrypts the connection and ensures that the server is legitimate, which prevents attackers from intercepting or modifying data in transit.

8. How does rate limiting enhance security in a web application, and how would you implement it?

   Rate limiting restricts the number of requests from a user, reducing brute-force attacks and abuse.

   ```
   // Example of rate limiting middleware in Express.js
   const rateLimit = require("express-rate-limit");
   const limiter = rateLimit({ windowMs: 15 * 60 * 1000, max: 100 }); // limit to 100 requests per 15 minutes
   app.use(limiter);
   ```

   This rate limiter restricts users to 100 requests per 15 minutes, reducing the likelihood of brute-force attacks and API misuse.

9. What is a security header, and how would you implement HTTP security headers to secure your application?

   Security headers instruct browsers on handling site content securely, reducing exposure to XSS, clickjacking, and other attacks.

   ```
   // Example in Express.js to set security headers
   app.use((req, res, next) => {
   res.setHeader("X-Content-Type-Options", "nosniff");
   res.setHeader("X-Frame-Options", "DENY");
   res.setHeader("Strict-Transport-Security", "max-age=63072000; includeSubDomains");
   next();
   });
   ```

   These headers prevent MIME sniffing, clickjacking, and ensure HTTPS for a site, strengthening web security.

10. Explain OAuth 2.0, and why it is often used for securing APIs in web applications.

    OAuth 2.0 is an authorization framework that allows applications to access resources on behalf of users without exposing their credentials, improving security for third-party access.

    ```
    POST /token HTTP/1.1
    Host: authorization-server.com
    Content-Type: application/x-www-form-urlencoded
    grant_type=authorization_code&code=AUTH_CODE&client_id=CLIENT_ID&client_secret=CLIENT_SECRET&redirect_uri=REDIRECT_URI
    ```

    OAuth 2.0 uses tokens to grant specific permissions, minimizing the need for sharing passwords across applications.

    learn more

RESTapi and GraphQL

1. How do you handle versioning in REST APIs, and why is it important?

    Versioning allows different versions of an API to exist simultaneously, so changes don't break existing clients. Versioning can be done via URL paths (`/api/v1/resource`), query parameters (`/resource?version=1`), or headers (`Accept: application/vnd.api.v1+json`).

    Using URL paths for versioning provides a clear structure and lets clients choose the API version without affecting others.

2. How would you handle pagination in a REST API? Give an example of a paginated response.

    Pagination limits the amount of data returned in a single response, often using query parameters like `page` and `limit` .

    ```
    GET /api/users?page=2&limit=10
    ```

    ```
    {
    "data": [ /* array of users */ ],
    "page": 2,
    "limit": 10,
    "total_pages": 5
    }
    ```

    The server responds with only a subset of users and includes metadata (`page` , `total_pages`) to help clients navigate through data.

3. Describe how you would implement JWT (JSON Web Token) authentication in a REST API.

    JWT authentication involves issuing a token upon login, which the client stores and sends with each request. The server verifies the token to authenticate the user.

    ```
    // Client-side: sending JWT in the Authorization header
    fetch('/api/secure-data', {
    headers: { 'Authorization': Bearer ${token} }
    });
    ```

    By including the token in the `Authorization` header, the server can validate the request and protect endpoints from unauthorized access.

    learn more

4. What is a resolver in GraphQL, and how does it work?

    A resolver is a function that fetches the data for a GraphQL query. Each field in a GraphQL schema has a corresponding resolver that determines how that field's data is retrieved

    ```
    const resolvers = {
    Query: {
    user: (parent, args, context) => {
    ```

```
return getUserById(
 args.id ); // Fetch user by ID
}
}
};
```

This resolver for a `user` query fetches the user data based on an ID, enabling GraphQL to map queries to specific data sources.

5. Explain the difference between queries, mutations, and subscriptions in GraphQL.

   - **Queries** are for reading data.
   - **Mutations** are for modifying data.
   - **Subscriptions** are for real-time updates when data changes.

6. How would you implement authentication in a GraphQL API?

   Authentication can be handled by checking a token in the request headers within the GraphQL context.

   ```
   // Setting up context for authentication
   const server = new ApolloServer({
   context: ({ req }) => {
   const token = req.headers.authorization || '';
   const user = getUserFromToken(token);
   return { user };
   }
   });
   ```

   By including `user` in the context, you can secure resolvers based on authentication status, allowing access only for authenticated users.

Trciky questions

HTML/CSS

1. **How do you vertically center an element inside a div with unknown height?**

   You can use Flexbox to center an element both horizontally and vertically inside a container, regardless of its height.

   ```
   .container {
   display: flex;
   align-items: center;  /* Vertical centering
   /
   justify-content: center;  /
    Horizontal centering  /
   height: 100vh; /
    optional: if you want full-screen height */
   }
   ```

   `align-items: center` aligns the element vertically, while `justify-content: center` centers it horizontally.

2. Explain the difference between `display: inline`, `display: inline-block`, and `display: block`.

   - `display: inline`: Elements only take up as much width as their content, and they don't allow vertical margin/padding.
   - `display: inline-block`: Like inline, but allows setting width, height, and vertical margins/padding.
   - `display: block`: Takes up the full width available and starts on a new line, allowing for width, height, and all margin/padding properties.

3. How can you create a responsive image gallery without using any media queries?

   Use CSS Grid with the `repeat` and `auto-fit` properties to create a responsive gallery.

   ```
   .gallery {
   display: grid;
   grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
   gap: 10px;
   }
   ```

   his setup automatically adjusts the number of columns based on available space, making the gallery responsive without media queries.

4. How can you make an element take up the full height of the viewport, even if its parent container has a limited height?

   Use `height: 100vh` to make the element span the full viewport height.

   ```
   .full-height {
   height: 100vh;
   }
   ```

   The `vh` unit represents a percentage of the viewport height, making it independent of parent containers.

5. How do you create a triangle shape in CSS?

   A triangle can be created by manipulating borders.

   ```
   .triangle {
   width: 0;
   height: 0;
   border-left: 50px solid transparent;
   border-right: 50px solid transparent;
   border-bottom: 100px solid blue;
   }
   ```

   By setting transparent borders and a non-transparent one, you can create a triangle shape with only CSS.

6. How do `position: relative` and `position: absolute` interact when used on a parent and child element?

   If a parent has `position: relative`, any child with `position: absolute` will be positioned relative to that parent, not the page.

   ```
   .parent {
   position: relative;
   }
   .child {
   position: absolute;
   top: 10px;
   left: 10px;
   }
   ```

   This positioning context is useful for placing elements precisely within a specific container.

7. How do you prevent an image from stretching when resizing the browser window in a responsive layout?

   Use `max-width: 100%; height: auto;` on the image element.

   ```
   img {
   max-width: 100%;
   height: auto;
   }
   ```

   This keeps the image proportional, allowing it to scale down responsively without stretching.

8. How can you create a gradient border on a button using CSS?

   Use a `background-image` gradient for the button's border, then clip it

   .`gradient-border {`
   ```
   border: 2px solid transparent;
   ```

```
border-image: linear-gradient(to right, red, blue) 1;
}
```

`border-image` allows for gradient borders by specifying the gradient as the border's source.

9. What is the `:not()` pseudo-class, and how would you use it to style all buttons except one with a specific class?

   The `:not()` selector applies styles to elements that do not match the specified selector.

   ```
   button:not(.special) {
   background-color: grey;
   }
   ```

   This selector makes it easy to apply styles to multiple elements while excluding specific ones.

   learn more

Javascript

1. What will be the output of the following code? Explain why

   ```
   console.log(0.1 + 0.2 === 0.3);
   ```

   The output will be `false`.

   Due to floating-point precision in JavaScript (and most other programming languages), `0.1 + 0.2` does not exactly equal `0.3`. The result is `0.30000000000000004`, so `0.1 + 0.2 === 0.3` evaluates to `false`.

2. How can you make the following code output `Hello, World!` using `setTimeout` with zero delay?

   ```
   console.log("Hello,");
   setTimeout(() => console.log("World!"), 0);
   ```

   ```
   console.log("Hello,");
   setTimeout(() => console.log("World!"), 0);
   ```

   Even with a delay of `0`, the function inside `setTimeout` is pushed to the event loop queue and will run after synchronous code. Hence, "Hello," is logged first, then "World!"

3. What will the following code output and why?

   ```
   var foo = "Hello";
   (function() {
   console.log(foo); // Output?
   var foo = "World";
   console.log(foo);
   })();
   ```

   The output will be:

   ```
   undefined
   World
   ```

   Due to variable hoisting, the `foo` variable inside the function is hoisted but uninitialized initially, making `console.log(foo)` output `undefined`. The second `console.log(foo)` outputs `"World"`.

4. How can you create a copy of an object in JavaScript?

   Method 1: Using `Object.assign()`:

   ```
   let obj = { a: 1, b: 2 };
   let copy = Object.assign({}, obj);
   ```

   Method 2: Using the spread operator:

```
let obj = { a: 1, b: 2 };
let copy = { ...obj };
```

Both methods create shallow copies of objects. For nested objects, a deep copy would be required, which can be done using `JSON.parse(JSON.stringify(obj))` or a library like Lodash.

5. What will be the output of the following code?

```
let a = [1, 2, 3];
let b = [1, 2, 3];
console.log(a == b);
console.log(a === b);
```

```
false
false
```

In JavaScript, arrays are compared by reference, not by value. Since `a` and `b` point to different objects in memory, they are not equal.

6. What will this code log to the console, and why?

```
let a = 10;
let b = (a++, a + 5);
console.log(a, b);
```

The output will be:

```
11 16
```

The comma operator evaluates each of its operands (left-to-right) and returns the value of the last operand. Here, `a++` increments `a` to `11`, and then `a + 5` is evaluated to `16` and assigned to `b`.

7. What will the following code output and why?

```
console.log(typeof null);
```

The output will be `object`.

In JavaScript, `null` is considered a primitive value but `typeof null` incorrectly returns `"object"` due to a bug in the original implementation of JavaScript.

React JS

1. What will be the output of the following code and why?

```
const [count, setCount] = React.useState(0);
```

```
function handleClick() {
setCount(count + 1);
setCount(count + 1);
}
```

```
return <button onClick={handleClick}>{count}</button>;
```

The output will increase by `1`, not `2`.

React batches state updates from event handlers for performance. Since `setCount(count + 1)` is called twice with the same `count` value, only one update takes effect. To increment twice, you should use the functional form:

```
setCount((prevCount) => prevCount + 1);
setCount((prevCount) => prevCount + 1);
```

2. How can you prevent a re-render in React when the component receives new props?

You can use `React.memo` for functional components to prevent re-renders when props haven't changed. For example:

```
const MyComponent = React.memo(({ data }) => {
// Component code
});
```

`React.memo` will only re-render the component if the props have changed. This is useful for optimization when rendering expensive components.

3. Why should you avoid using `index` as a key in a list?

   Using `index` as a key can lead to performance issues and incorrect behavior when the list order changes, as React relies on keys to determine which items have changed.

   If the order changes or new items are added, using `index` may cause React to re-render components unnecessarily. Instead, use unique identifiers when possible

4. What will the following code output and why?

   ```
   const [count, setCount] = React.useState(0);
   ```

   ```
   React.useEffect(() => {
   console.log(count);
   }, [count]);
   ```

   ```
   setCount(5);
   ```

   This code will throw an error: "Maximum update depth exceeded.”

   `setCount(5)` is invoked outside of an event handler or a lifecycle method, causing an infinite re-render loop. `setCount` should only be called within event handlers or effects

5. What is the purpose of `React.StrictMode`?

   `React.StrictMode` is a tool for highlighting potential issues in an application. It helps identify unsafe lifecycle methods, legacy API usage, and unexpected side effects.

   `StrictMode` doesn’t render anything visible to the user. It’s used in development to help ensure the code adheres to best practices and prepares for future React releases.

6. What’s the difference between `useMemo` and `useCallback`?

   - **`useMemo`**: Memoizes a computed value. Useful for caching the result of an expensive calculation.
   - **`useCallback`**: Memoizes a function, which is useful for optimizing callbacks in dependency arrays of other hooks or for `React.memo`.

   `useMemo` is for caching computed values, while `useCallback` is for caching functions to avoid unnecessary re-renders of child components that depend on the function reference.

7. How do closures affect hooks in React? Explain with an example.

   Closures can lead to stale values if a hook references a state variable at the time of its definition and that value doesn't update with re-renders.

   ```
   const [count, setCount] = React.useState(0);
   ```

   ```
   function handleClick() {
   setTimeout(() => {
   alert(count); // May show stale value
   }, 1000);
   }
   ```

   ```
   return <button onClick={handleClick}>Show Count</button>;
   ```

   The `alert` might show a stale `count` value if the component re-renders before the timeout is executed. Using `setCount((prevCount) => prevCount + 1)` or capturing `count` in a closure can help avoid this.

8. What are potential issues of using inline functions in the `render` method or JSX?

Inline functions create a new function instance on every render, which can cause performance issues and unnecessary re-renders of child components.

Each time the component re-renders, a new function reference is created, which can prevent `React.memo` or `PureComponent` from optimizing re-renders. Use `useCallback` to avoid this problem.

9. What will the following code output, and why?

```
function Parent() {
const [count, setCount] = React.useState(0);

return (
<>
<button onClick={() => setCount(count + 1)}>Increment</button>
<Child count={count} />
</>
);
}

const Child = React.memo(({ count }) => {
console.log("Child rendered");
return <div>{count}</div>;
});
```

`Child rendered` will be logged every time `count` changes.

`React.memo` prevents `Child` from re-rendering if `count` hasn't changed, but in this case, `count` is a primitive value, so `Child` will re-render each time `count` changes

Django

1. What is the difference between `null=True` and `blank=True` in Django models?

   `null=True` allows database fields to store `NULL` values, while `blank=True` allows form fields to be left empty in forms.

   `null=True` is for database schema, allowing `NULL` in the database, whereas `blank=True` is for validation in Django forms. A field with `null=True, blank=True` can have both empty form validation and store `NULL` in the database.

2. How does Django handle database migrations when you rename a model field?

   Django creates an `AlterField` migration, which may drop the old field and create a new one with the updated name.

   Renaming a field doesn't automatically migrate data or retain previous data. To retain data during renaming, use the `RenameField` operation or create a custom migration.

3. What does the `@transaction.atomic` decorator do, and when would you use it?

   `@transaction.atomic` ensures that the entire block of code within the function executes as a single database transaction. If any error occurs, the entire transaction is rolled back.

   Use `@transaction.atomic` when you need to ensure that a set of database operations completes fully or not at all. This prevents partial updates in case of errors, ensuring data integrity.

4. Why should you avoid using the `save()` method on Django QuerySets, and what's the alternative?

   Using `save()` on QuerySets is inefficient and doesn't execute in a single SQL query, causing individual database writes for each object.

   Use `update()` on QuerySets, which performs a bulk update with a single SQL query.

5. How does Django prevent SQL injection, and what is the role of ORM in this?

Django ORM uses parameterized queries to prevent SQL injection, meaning that it doesn't allow direct concatenation of user input into SQL commands.

By using parameterized queries, Django automatically escapes values, ensuring that malicious SQL code can't be injected. Always use ORM methods rather than raw SQL for safety.

6. What are Django signals, and when should you avoid using them?

   Django signals allow certain actions to trigger specific functions, such as updating a user profile when a user is created.

   You need a clean, easily understandable code flow. Overusing signals can make the code hard to trace and debug, so it's better to use them sparingly and only for clear decoupling needs.

7. How do you handle circular imports in Django?

   Use lazy loading with `from <app>.models import Model` inside functions, or import models using `get_model` from `django.apps`.

   Circular imports occur when two modules depend on each other. To resolve this, delay imports by placing them inside functions or use `django.apps.get_model('app_name', 'ModelName')` for models.

8. Why should you avoid using `HttpResponseRedirect` directly, and what's the safer alternative?

   `HttpResponseRedirect` doesn't handle URL reversing, which can lead to hard-coded URLs and make code changes more difficult.

   Use `redirect('view_name')` or `reverse('view_name')`, which rely on the view's name and allow URLs to change without affecting code.

9. In Django, why might `get_object_or_404` throw an exception if used within a custom `ModelManager` method?

   If `get_object_or_404` is used inside a custom manager, the manager itself can't handle `404` responses, causing the view to receive an unhandled exception instead.

   `get_object_or_404` is intended for views where a `404` response is meaningful. If needed in a manager, manually handle exceptions or avoid using it and handle `None` or `DoesNotExist` errors instead.

MongoDB & SQL

1. How does `JOIN` differ from `UNION` in SQL?

   `JOIN` combines columns from different tables based on a related column, resulting in a wider table with additional columns. `UNION` combines rows from two queries with identical column structures into a single result set, stacking rows vertically.

2. What is the purpose of the `WITH` clause (CTE) in SQL, and why is it useful

   The `WITH` clause (Common Table Expression or CTE) allows the creation of a temporary result set that can be referenced multiple times in the main query. It's useful for improving readability, simplifying complex queries, and reusing subquery logic.

3. How can you optimize a slow `JOIN` query

   - **Index the joining columns** to speed up lookup.
   - **Use INNER JOIN** only when necessary, as it is generally faster than other joins.

- **Limit unnecessary columns** in your `SELECT` statement.
- **Avoid functions on columns** in the `JOIN` condition, as this can prevent index usage

4. How would you find the second-highest salary from an employee table?

   Use a subquery or the `ROW_NUMBER()` window function.

   ```
   SELECT MAX(salary) AS second_highest_salary
   FROM employees
   WHERE salary < (SELECT MAX(salary) FROM employees);
   ```

   This query finds the highest salary below the maximum salary.

5. What's the difference between `find()` and `aggregate()` in MongoDB?

   `find()` retrieves documents based on simple queries, ideal for basic CRUD operations. `aggregate()` is used for more complex operations like transformations, grouping, and calculations using stages in the aggregation pipeline.

   learn more

6. How does MongoDB handle transactions differently than SQL databases, and when should you use them?

   MongoDB supports multi-document transactions, but unlike SQL, transactions are not native to all collections (especially on sharded clusters). Use transactions only when you need ACID compliance across multiple documents, as MongoDB is optimized for non-transactional operations by default.

7. How can you avoid performance issues when querying large collections in MongoDB

   Use **indexes** on frequently queried fields, avoid **$regex** on large collections, **limit** the number of fields retrieved with `projection`, and **avoid unbounded queries** which can scan the entire collection. Also, use **pagination** with `skip()` and `limit()`

8. How would you find duplicate records in a MongoDB collection?

   Use `$group` with `$count` in the aggregation pipeline to identify duplicate records.

   ```
   db.collection.aggregate([
   { $group: { _id: "$field_name", count: { $sum: 1 } } },
   { $match: { count: { $gt: 1 } } }
   ])
   ```

   This pipeline groups documents by `field_name`, counts them, and filters those with a count greater than 1, indicating duplicates.

Web Security

1. What is the difference between `HTTPS` and `HTTP` and why isn't HTTPS foolproof?

   `HTTP` (Hypertext Transfer Protocol) is an unsecured protocol for transmitting data over the internet, while `HTTPS` (HTTP Secure) uses TLS/SSL to encrypt data, ensuring confidentiality and integrity during transmission. However, HTTPS is not foolproof because it only secures data in transit, not data at rest. Attacks like SSL stripping and man-in-the-middle can still occur if proper configurations and updated protocols are not used.

2. How does Cross-Site Scripting (XSS) differ from Cross-Site Request Forgery (CSRF)?

   XSS exploits a vulnerability in a website to inject malicious scripts, typically affecting other users by running the script in their browsers. CSRF, on the other hand, tricks a user's browser into performing unwanted actions on a different site where they are authenticated. XSS affects the client directly, while CSRF takes advantage of the user's authenticated session.

3. What is SQL injection, and why do parameterized queries prevent it?

SQL injection is a technique where attackers inject malicious SQL code into queries, exploiting a lack of input sanitization. Parameterized queries separate the SQL code from the input data, meaning user input is treated as data only, not executable code, preventing the query structure from being modified by user input.

4. What is `Content Security Policy (CSP)`, and how does it prevent certain attacks?

   CSP is an HTTP header that restricts sources for various types of content (like scripts, images, or styles) in a web application. It helps prevent attacks such as XSS by allowing only approved sources to load scripts or styles. If an attacker injects malicious scripts, they will be blocked by the browser if they do not match the CSP policy.

5. How does a SameSite cookie attribute help prevent CSRF attacks, and what are its limitations?

   The `SameSite` cookie attribute restricts cookies from being sent with cross-site requests. Setting it to `SameSite=Strict` or `SameSite=Lax` helps prevent CSRF by ensuring that cookies are only sent with requests initiated from the same site. However, it's limited because it does not protect against attacks that exploit vulnerabilities in other areas, such as compromised user accounts.

6. Why is hashing alone not sufficient for storing user passwords securely?

   Hashing alone is inadequate because many hashing algorithms are fast, allowing attackers to perform brute-force or rainbow table attacks. Instead, use a salted hash with a slow hashing algorithm like `bcrypt` or `Argon2`, which introduces a unique salt for each password and makes brute-force attacks computationally expensive.

7. What is the principle of least privilege, and how does it apply to web security?

   The principle of least privilege means giving users and applications the minimum access needed to perform their tasks. In web security, this reduces the potential impact of a compromise, as attackers cannot access resources or execute commands beyond those permitted for the compromised account or process.

   learn more

8. Explain how clickjacking works and one way to prevent it.

   Clickjacking is an attack where a malicious site overlays a transparent iframe over a legitimate site, tricking users into clicking on unintended elements. To prevent clickjacking, websites can use the `X-Frame-Options` HTTP header (set to `DENY` or `SAMEORIGIN`) or `Content-Security-Policy: frame-ancestors 'none'` to control which sites are allowed to frame their content.

9. What is a `Security Misconfiguration` and give an example.

   A security misconfiguration occurs when an application, database, or server is improperly configured, leaving it vulnerable to attacks. An example is leaving default usernames and passwords on a database, which attackers can exploit to gain unauthorized access.

   learn more

10. Why should sensitive data not be stored in local storage in browsers?

    Local storage is not a secure place for sensitive data because it can be accessed by any JavaScript code running on the page, including potentially malicious scripts injected via XSS attacks. Instead, sensitive data should be stored in secure HTTP-only cookies or in-memory for the session, to ensure it's not accessible by JavaScript.

    learn more

REST and GraphQL

1. What is the difference between `PUT` and `PATCH` methods?

`PUT` is used to update a resource by replacing it entirely, while `PATCH` is used for partial updates. For instance, with `PUT`, you send the entire updated object, but with `PATCH`, only the fields you wish to update are required.

learn more

2. What are the benefits and potential risks of using JSON Web Tokens (JWT) for authentication in REST APIs?

   Benefits include stateless sessions and scalability, as JWTs contain user information and can be verified without database calls. However, risks involve token theft, which allows attackers to impersonate users, so securing token storage and transmission is critical.

3. Why is `OPTIONS` method used in REST APIs, and when is it typically seen?

   The `OPTIONS` method is used in CORS preflight requests to check server permissions on allowed HTTP methods, headers, and origins before sending the actual request. It's common in browsers when making cross-origin API calls.

   learn more

4. What is `HATEOAS`, and why is it important in REST API design

   `HATEOAS` (Hypermedia as the Engine of Application State) allows REST APIs to include links in responses, guiding clients to available actions without hardcoding endpoint structures. It promotes discoverability and reduces client-server coupling

   learn more

5. Explain idempotency in REST APIs with examples

   Idempotency ensures that making the same request multiple times results in the same effect. For example, `GET` and `DELETE` are idempotent as calling them multiple times doesn't change the outcome, whereas repeated `POST` requests may create multiple entries unless explicitly handled.

   learn more

6. How do you handle file uploads in REST APIs, and what security concerns are associated?

   File uploads can be handled by setting the `Content-Type` to `multipart/form-data` and storing files in secure directories or cloud storage. Security concerns include validating file types, preventing large file uploads, and scanning for malware

7. How does GraphQL handle over-fetching and under-fetching compared to REST

   GraphQL allows clients to specify exactly what data they need, addressing over-fetching and under-fetching issues in REST, where responses often include either too much or too little data. This makes GraphQL more efficient in data retrieval.

8. What is N+1 query problem in GraphQL, and how do you solve it?

   The N+1 problem occurs when GraphQL makes multiple database calls for nested queries, which is inefficient. Solutions include using a DataLoader or batching to group queries, reducing the number of database calls.

   learn more

9. Why should you use field aliasing in GraphQL, and provide an example

   Field aliasing allows renaming fields in a query to avoid conflicts or add clarity

   ```
   query {
   user: getUser(id: 1) { name }
   admin: getUser(id: 2) { name }
   }
   ```

   This query renames fields to differentiate `user` and `admin` data in a single response.

10. How can you implement authorization in GraphQL

Authorization in GraphQL can be implemented at the resolver level, using middleware or directives. This enables role-based access, ensuring users can only access or modify allowed resources.

learn more

11. Describe the purpose of GraphQL schema stitching

    Schema stitching combines multiple GraphQL schemas into one, allowing clients to access data from various sources with a single unified schema. It's helpful for integrating data from microservices or third-party APIs.

    learn more

12. What is introspection in GraphQL, and why can it be a security risk

    Introspection allows clients to explore the GraphQL schema, which helps in development but can expose the API structure to attackers. Disabling introspection in production helps mitigate this risk.

    learn more

13. How does caching work in GraphQL, and why is it challenging

    Caching in GraphQL is challenging because requests are more dynamic than in REST, with clients specifying different fields each time. Techniques like persisted queries, CDN caching, and client-side caching libraries (e.g., Apollo Client) help overcome this challenge.

    learn more

Interview based questions

HTML/CSS

1. Describe how you would implement a responsive design layout for a website. What techniques would you use to ensure it works well across various screen sizes?

   Responsive design is essential for creating a seamless experience across devices, from desktops to mobile phones

   CSS Media Queries to apply different styles based on screen size. For instance

   ```
   @media (max-width: 768px) {
   .container {
   width: 100%;
   padding: 20px;
   }
   }
   ```

   Flexible Grids with a CSS framework like Bootstrap or using CSS Grid and Flexbox, which allow elements to adjust based on available space.

   Viewport Units (e.g., `vw`, `vh`) to size elements relative to the viewport for fluid scaling

   Responsive Images with `srcset` or CSS properties like `object-fit` for different image sizes to optimize performance and appearance across devices.

   Rem or Em Units instead of pixels for scalable typography and spacing that adjusts with device and user settings.

   learn more

2. Explain how you would optimize CSS for performance on a large-scale website.

   Optimizing CSS for performance on large-scale websites involves several steps:

- **Minification and Compression**: Minify CSS files to reduce file size, and use GZIP compression for faster loading.
- **Modular CSS Structure**: Use CSS methodologies like BEM (Block Element Modifier) to create reusable, maintainable, and predictable class structures.
- **Avoiding Inline Styles**: Keep styles in separate CSS files or apply them selectively in a `<style>` tag within the `<head>` for critical CSS.
- **CSS Preprocessors** (like SASS or LESS): To organize, nest, and reuse CSS efficiently, resulting in smaller, more maintainable code.
- **Load Critical CSS First**: Use critical CSS in the `<head>` for above-the-fold content and load non-critical CSS asynchronously to improve perceived page load times.
- **Avoid Expensive Selectors**: Prefer classes over descendant selectors and limit the depth of selectors to reduce CSS processing times.

learn more

3. How would you create a custom-styled checkbox or radio button using CSS?

   Styling native checkboxes and radio buttons requires hiding the default input and replacing it with a custom design:

```
<label class="custom-checkbox">
<input type="checkbox">
<span class="checkmark"></span>
Agree to Terms
</label>
```

```
.custom-checkbox input[type="checkbox"] {
display: none; /* Hide the native checkbox */
}
.custom-checkbox .checkmark {
width: 20px;
height: 20px;
border: 2px solid #000;
display: inline-block;
border-radius: 4px;
position: relative;
}
.custom-checkbox input[type="checkbox"]:checked + .checkmark {
background-color: #4CAF50;
}
```

   The native checkbox is hidden using `display: none;`, and the custom `span` element styled to resemble a checkbox. When checked, the sibling `.checkmark` changes background color. This provides a fully custom-styled checkbox while maintaining accessibility by keeping the input in the DOM.

javascript

1. How would you implement a debounce function in JavaScript, and when would you use it?

   A debounce function limits the rate at which a function can fire. It is especially useful in scenarios like search boxes, where it prevents the function from executing every time a user types a character, instead only executing after the user has stopped typing for a specified amount of time. Here’s an example implementation:

```
function debounce(func, delay) {
let timeout;
```

```
return function(...args) {
clearTimeout(timeout);
timeout = setTimeout(() => func.apply(this, args), delay);
};
}
```

The `debounce` function returns a new function that clears any previous `timeout` and sets a new one. Only after the specified `delay` time has passed without new calls will `func` execute. This improves performance by reducing unnecessary function calls, which is valuable in handling events like `resize`, `scroll`, and `input`.

2. How would you deep clone an object in JavaScript? What are some of the pitfalls of using `JSON.parse(JSON.stringify(obj))` for this purpose?

   A deep clone means copying an object with all nested properties completely independent of the original. There are several ways to achieve deep cloning in JavaScript:

   Using `JSON.parse(JSON.stringify(obj))`:

   ```
   const original = { a: 1, b: { c: 2 } };
   const clone = JSON.parse(JSON.stringify(original));
   ```

   Pitfall: This approach doesn't handle functions, `Date` objects, `RegExp`, `undefined`, or circular references correctly.

   Using recursion or libraries

   ```
   function deepClone(obj) {
   if (obj === null || typeof obj !== "object") return obj;
   if (obj instanceof Date) return new Date(obj.getTime());
   if (obj instanceof Array) return obj.map(item => deepClone(item));
   const copy = {};
   for (let key in obj) {
   copy[key] = deepClone(obj[key]);
   }
   return copy;
   }
   ```

   This recursive function is more reliable than `JSON.parse(JSON.stringify(obj))` for cloning complex data types, but using a library like Lodash (`_.cloneDeep`) is a common industry approach as it handles edge cases efficiently.

React JS

1. How would you manage complex state in a React application with multiple components needing access to shared data?

   Managing complex state across components can be challenging. Here are common approaches:

   - **Context API**: Provides a way to pass data through the component tree without props drilling. Suitable for global or semi-global state that changes infrequently.
   - **State Management Libraries (e.g., Redux, Zustand, Recoil)**: For more complex or frequently changing state, libraries like Redux or Recoil help manage shared state, especially if components need access to and can modify shared data.
   - **useReducer Hook**: Useful for managing local component state that's complex, such as forms or multi-step workflows.
   - **Example**:

     ```
     javascript
     Copy code
     ```

```jsx
import React, { createContext, useContext, useReducer } from 'react';

const CountContext = createContext();

function countReducer(state, action) {
    switch (action.type) {
        case 'increment':
            return { count: state.count + 1 };
        case 'decrement':
            return { count: state.count - 1 };
        default:
            throw new Error();
    }
}

export function CountProvider({ children }) {
    const [state, dispatch] = useReducer(countReducer, { count: 0 });
    return (
        <CountContext.Provider value={{ state, dispatch }}>
            {children}
        </CountContext.Provider>
    );
}

export function useCount() {
    return useContext(CountContext);
}
```

In this example, `CountProvider` and `useCount` make it easy to share state between components while maintaining predictable state changes via `dispatch`.

2. Explain how React's Reconciliation process works and how it impacts rendering performance.

    - React's Reconciliation process is a core algorithm responsible for determining which parts of the DOM need to be updated when the application's state or props change. It uses a virtual DOM to perform a "diffing" process, comparing the current and previous DOM states, then applying the minimal number of changes needed to update the real DOM.

      Optimizations React employs:

        - **Keyed elements**: Helps React efficiently track items in lists, so only elements that have changed are updated.
        - **Pure components and memoization**: `React.memo` and `PureComponent` prevent unnecessary re-renders if the props or state haven't changed.

    - **Performance Impact**: By minimizing direct DOM manipulation (which is costly), the Reconciliation process allows React to perform updates quickly and efficiently, improving rendering performance. Proper use of keys in lists and avoiding unnecessary re-renders are best practices to leverage React's Reconciliation efficiency.

3. What are Higher-Order Components (HOCs) in React, and how would you implement one?

A Higher-Order Component (HOC) is a function that takes a component and returns a new component with additional functionality. HOCs are commonly used to add cross-cutting concerns, such as authentication or data fetching, to components without modifying them directly.

Example of an HOC:

```
function withLogging(WrappedComponent) {
    return function EnhancedComponent(props) {
        useEffect(() => {
            console.log("Component mounted");
            return () => console.log("Component unmounted");
        }, []);

        return <WrappedComponent {...props} />;
    };
}

function MyComponent() {
    return <div>Hello World</div>;
}

export default withLogging(MyComponent);
```

In this example, `withLogging` is an HOC that adds logging behavior when the component mounts and unmounts. HOCs help apply reusable logic, but should be used with caution to avoid wrapper hell (too many nested HOCs) and reduced readability.

Django

1. Describe the Django request-response cycle. How does Django handle a request from the client to the server and return a response?

   The Django request-response cycle begins when a client sends an HTTP request to the server. Django handles this request through a series of steps:

   1. **URL Routing**: Django's URL dispatcher (`urls.py`) matches the request's URL to a view based on the URL patterns defined in the `urls.py` file.
   2. **View Execution**: Once a matching URL is found, Django calls the corresponding view function. This view processes the request, often interacting with the database or other services.
   3. **Response Generation**: The view function returns an HTTP response, either using `HttpResponse`, `JsonResponse`, or by rendering a template.
   4. **Middleware**: Throughout this process, middleware functions can intercept and modify requests and responses, performing tasks like authentication, logging, and error handling.

   Understanding the request-response cycle is essential for debugging and optimizing Django applications, as it affects how data flows and where middleware or caching can be applied for efficiency.

2. How would you handle file uploads in Django and secure access to these files?

- File uploads in Django are handled using `FileField` or `ImageField` in a model and setting up a form to allow users to upload files. The file is saved to the location specified by `MEDIA_ROOT` in settings. To secure these files:
  - **Control Access**: Use view-based permissions to restrict access to files, or store sensitive files outside the publicly accessible `MEDIA_ROOT` and serve them via a view with access checks.
  - **Limit File Types and Size**: Validate files to allow only specific types (e.g., images or PDFs) and limit the file size during upload to prevent storage abuse.
  - **Sanitize File Names**: Use Django's built-in `django.utils.text.get_valid_filename` to sanitize file names, preventing security risks associated with special characters.
- **Example**:

```
def protected_file_view(request, file_name):
    if request.user.is_authenticated:
        file_path = os.path.join(settings.MEDIA_ROOT, 'protected', file_nam
e)
        with open(file_path, 'rb') as file:
            response = HttpResponse(file.read(), content_type="application/
force-download")
            response['Content-Disposition'] = f'attachment; filename={file_
name}'
            return response
    else:
        return HttpResponseForbidden("Not authorized")
```

Proper file handling and access control are critical to securing a Django application, preventing unauthorized access to files, and safeguarding server resources.

3. Explain Django's ORM and how you would structure a many-to-many relationship with an example.

   Django's Object-Relational Mapping (ORM) translates Python classes into database tables, making it easier to interact with the database without writing SQL. A many-to-many relationship in Django can be defined using `ManyToManyField`.

   Example: Suppose we have an app to manage courses and students where each student can enroll in multiple courses, and each course can have multiple students.

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    courses = models.ManyToManyField('Course', related_name='students')

class Course(models.Model):
    title = models.CharField(max_length=100)
```

- In this setup, Django will automatically create an intermediary table to manage the many-to-many relationship. The `related_name` attribute allows you to access students from a course (e.g., `course.students.all()`).

Using Django's ORM simplifies creating and querying complex relationships, which can be useful for performance optimizations and readability in projects with interconnected data.

4. How can you implement custom validation in Django forms, and why is it important?

   Custom validation in Django forms ensures that the data submitted by users meets specific criteria beyond the default validation rules. This is crucial for enforcing data integrity and preventing incorrect data from entering the system.

   Example: Suppose we want to add a custom validator to ensure that a username has at least 8 characters.

```
from django import forms

class RegistrationForm(forms.Form):
    username = forms.CharField(max_length=150)
    password = forms.CharField(widget=forms.PasswordInput)

    def clean_username(self):
        username = self.cleaned_data.get("username")
        if len(username) < 8:
            raise forms.ValidationError("Username must be at least 8 character
s long.")
        return username
```

Here, the `clean_username` method is a custom validator that raises a `ValidationError` if the username is too short. Django's form validation is crucial for enforcing rules, as it allows us to catch errors at the form level before saving data to the database.

5. Describe how Django's authentication system works. How would you create a custom user model to add additional fields?

   Django's authentication system provides a robust framework for handling user login, logout, and session management. Django includes a default `User` model with basic fields like `username`, `password`, and `email`. However, in many cases, applications require additional fields or modifications.

   Creating a Custom User Model: To add custom fields, you can create a custom user model by subclassing `AbstractUser` or `AbstractBaseUser`.

```
from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    age = models.PositiveIntegerField(null=True, blank=True)
    phone_number = models.CharField(max_length=15, blank=True, null=True)
```

- **Setting the Custom Model**: You then set the `AUTH_USER_MODEL` in `settings.py`:

```
AUTH_USER_MODEL = 'myapp.CustomUser'
```

Creating a custom user model is essential when additional fields or different validation logic are required for user profiles. `AbstractUser` is a straightforward choice for custom fields, while `AbstractBaseUser` offers more customization but requires implementing core methods like `get_full_name` and `get_short_name`.

MongoDB & SQL

1. When would you choose MongoDB over SQL, and vice versa, in terms of application requirements?
   - **MongoDB**: Choose MongoDB for applications that require flexible schemas, quick development cycles, and handling of semi-structured data. Examples include content management systems, real-time analytics, and applications with rapidly changing requirements. MongoDB is schema-less, which means documents can have different fields, making it ideal for projects with dynamic data structures.
   - **SQL**: Choose SQL databases for applications that require strict data integrity, complex queries, and transactional support (ACID compliance). Examples include financial applications, e-commerce systems, and applications requiring data consistency and relationship enforcement. SQL databases enforce a schema, so every row in a table follows the same structure, making them suitable for applications with well-defined data.
2. How would you model a one-to-many relationship in SQL vs. MongoDB?

   In SQL

   : A one-to-many relationship can be modeled using foreign keys. For example, in an e-commerce database, a

   ```
   customer
   ```

   table might have a one-to-many relationship with an

   ```
   orders
   ```

   table. Each

   ```
   order
   ```

   would store a foreign key reference to the

   ```
   customer
   ```

   that placed it.

   ```
   CREATE TABLE Customer (
       customer_id INT PRIMARY KEY,
       name VARCHAR(100)
   ```

```
);

CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    product_name VARCHAR(100),
    FOREIGN KEY (customer_id) REFERENCES Customer(customer_id)
);
```

In MongoDB: One-to-many relationships can be modeled by embedding documents or by using references. If the data is highly related and often queried together, embedding is efficient. For example:

```
{
"_id": 1,
"name": "John Doe",
"orders": [
{"order_id": 101, "product_name": "Laptop"},
{"order_id": 102, "product_name": "Phone"}
]
}
```

Embedding in MongoDB is efficient for frequently accessed nested data, while SQL's normalized structure with foreign keys is preferred for enforcing strict relationships and avoiding redundancy.

3. How can you migrate a relational database structure to MongoDB? Describe the key changes you would make to the schema design

   Migrating from a relational database to MongoDB requires denormalizing the schema, as MongoDB doesn't enforce strict table relationships:

   - **Identify Relationships**: Analyze your SQL schema to understand the one-to-many and many-to-many relationships.
   - **Embed vs. Reference**: Use embedded documents for tightly coupled data (e.g., user profiles with addresses), and references for loosely coupled data (e.g., customers and orders).
   - **Flatten Structure**: Replace joins with embedded documents or arrays to avoid complex querying. For example, an `Order` table with multiple items may become an order document with an `items` array.
   - SQL Structure:

     ```
     Customer Table: customer_id, name
     Order Table: order_id, customer_id, order_details
     ```

   - MongoDB Structure:

     ```
     {
       "_id": "customer_id",
       "name": "Customer Name",
       "orders": [
           {"order_id": "1", "order_details": "Details"},
           {"order_id": "2", "order_details": "Details"}
     ```

```
  ]
}
```

MongoDB prioritizes performance and flexibility over data normalization, so denormalizing data to fit MongoDB's document-based model optimizes data retrieval.

Web security

1. Describe Cross-Site Request Forgery (CSRF) and how modern web applications defend against it.

   Cross-Site Request Forgery (CSRF) is an attack that tricks users into performing actions on a web application where they are authenticated, without their consent.

   - **Prevention**:
     - **CSRF Tokens**: Add unique, random tokens to every form that the server validates upon receiving a request, ensuring the request originated from the legitimate user.
     - **SameSite Cookies**: Set cookies to `SameSite=Strict` or `SameSite=Lax` to prevent them from being sent with cross-site requests.
     - **Double Submit Cookie**: In cases where CSRF tokens are challenging to implement, a "double submit" cookie can be used, where both a CSRF token in the cookie and a request body/URL parameter must match for validation.

2. Explain HTTPS and how it contributes to web security. What role does SSL/TLS play in HTTPS?

   HTTPS (Hypertext Transfer Protocol Secure) is an extension of HTTP that encrypts data between the user's browser and the web server, preventing data interception or tampering.

   - **SSL/TLS**: HTTPS relies on SSL (Secure Sockets Layer) or TLS (Transport Layer Security) protocols to establish a secure, encrypted connection. SSL/TLS encrypts data to:
     - **Ensure Confidentiality**: Encrypts the data, making it readable only by the intended recipient.
     - **Maintain Data Integrity**: Prevents data alteration during transmission.
     - **Authenticate Server Identity**: Verifies the server's identity through digital certificates, protecting against man-in-the-middle attacks.
   - **Industry Best Practices**: Ensure proper SSL/TLS configuration, including using strong ciphers and disabling outdated protocols like SSL 2.0 and SSL 3.0.

3. What is a Man-in-the-Middle (MITM) attack, and what are some ways to defend against it?

   A Man-in-the-Middle (MITM) attack occurs when an attacker secretly intercepts and possibly alters the communication between two parties who believe they are directly communicating with each other.

   - **Defenses**:
     - **Use HTTPS**: Enforcing HTTPS ensures that all data transmitted between client and server is encrypted, preventing MITM attacks on data in transit.
     - **Public Key Pinning**: Prevents attackers from using fraudulent certificates by "pinning" trusted public keys or certificates to the application.

- **DNS Security Extensions (DNSSEC)**: Protects against DNS spoofing by signing DNS responses, ensuring that users are directed to the correct servers.
- **End-to-End Encryption**: Encrypt sensitive data end-to-end so that only the sender and receiver can decrypt it, even if an MITM intercepts the communication.

REST and GraphQL

1. When designing an API for a large, complex application, what factors would influence your decision to choose REST over GraphQL, or vice versa?
   - **REST** is often preferred for:
     - **Simplicity and predictability**: REST endpoints are well-defined and follow standard HTTP methods, which makes it easy to understand and work with.
     - **Caching**: REST's structure enables straightforward caching mechanisms for commonly requested resources.
     - **Statelessness**: REST is typically stateless, making it more suitable for applications where state does not need to persist between calls.
     - **Mature ecosystem**: Tools, best practices, and community support are robust due to REST's long-standing use.
   - **GraphQL** is often chosen for:
     - **Flexible querying**: Clients can request only the data they need, reducing over-fetching and under-fetching of data, which is useful for complex UIs.
     - **Multiple resource fetching**: GraphQL allows clients to fetch related resources in a single request, which minimizes round trips to the server.
     - **Versionless API**: GraphQL's schema evolution allows for updates without versioning, making it ideal for iterative application development.
     - **Real-time updates**: GraphQL supports subscriptions, which enable real-time data updates more efficiently than REST.
2. How would you handle authentication and authorization differently in a REST API versus a GraphQL API?
   - **REST API**:
     - **Token-based Authentication**: Typically, REST APIs use tokens (e.g., JWTs) passed in the `Authorization` header with each request.
     - **Role-based Access Control (RBAC)**: Different endpoints can enforce various roles by checking the token's role claims, allowing granular access control at the route level.
     - **Endpoint-specific Permissions**: Each endpoint can enforce its own permissions, depending on the user's role or group.
   - **GraphQL API**:
     - **Single Endpoint Authentication**: Since GraphQL uses a single endpoint, authentication must be managed within resolvers to validate the token for each query or mutation.
     - **Field-level Authorization**: Authorization logic can be applied at the resolver level, allowing fine-grained access control on individual fields, which is beneficial for complex data structures.

- **Middleware**: Custom middleware can be used to validate tokens before executing resolvers, reducing redundancy in handling authentication.

3. In a performance-critical application, how would you address the over-fetching and under-fetching issues in REST, and why might GraphQL still present performance challenges?
   - **In REST**:
     - **Pagination and filtering**: Implement pagination, sorting, and filtering mechanisms to reduce data transfer and prevent over-fetching.
     - **Resource-based endpoints**: Design endpoints that align with the client's data needs, such as combining related resources into a single endpoint for specific use cases.
     - **Caching**: Use caching layers (e.g., CDN or HTTP cache) to speed up responses for commonly requested data.
   - **In GraphQL**:
     - **Over-fetching risk**: Although GraphQL minimizes over-fetching by allowing selective data querying, it can cause performance issues if clients request deeply nested or complex queries that require multiple database calls.
     - **Rate limiting and query complexity analysis**: Implement tools to limit query depth and complexity, preventing excessively expensive operations.
     - **Caching**: GraphQL doesn't natively support caching like REST does, so external caching strategies (e.g., caching specific fields or query responses) are necessary to maintain performance.
4. How would you implement real-time data synchronization in both REST and GraphQL for a collaborative application where multiple users need live updates?
   - **In REST**:
     - **Polling**: Clients could poll specific endpoints periodically for updates, though this can be inefficient.
     - **WebSockets**: For real-time capabilities, WebSocket connections can be implemented separately from REST, allowing bi-directional communication without continuous HTTP requests.
     - **Server-Sent Events (SSE)**: In some cases, SSE can be used to push real-time updates from the server to the client in a unidirectional stream.
   - **In GraphQL**:
     - **Subscriptions**: GraphQL supports subscriptions, allowing clients to subscribe to specific changes in data and receive real-time updates.
     - **WebSocket Protocol**: Subscriptions are usually implemented over WebSocket, providing a persistent connection for instantaneous data updates.
     - **Integration with Resolver Logic**: Real-time updates can be integrated directly with mutation resolvers, enabling efficient synchronization between clients whenever a mutation occurs.
5. Describe an approach for error handling in REST and GraphQL APIs, especially when dealing with complex requests that involve multiple data sources.
   - **In REST**:
     - **Standard HTTP Status Codes**: Use standardized HTTP status codes (e.g., `404`, `500`, `403`) to indicate error types at the endpoint level.

    - **Detailed Error Responses**: Include specific error messages in JSON format in the response body, detailing the cause and possible remedies for client-side errors.
    - **Batch Operations**: For requests that involve multiple resources, partial success and failure details should be included in the response to clarify which operations succeeded or failed.
- **In GraphQL**:
    - **Error Object in Responses**: GraphQL returns errors as part of the response payload, allowing both data and error messages in a single response.
    - **Partial Success**: GraphQL can deliver partial results alongside error details for fields that failed, which is helpful in complex, multi-field queries.
    - **Custom Error Handling**: Custom error classes can be implemented at the resolver level to provide detailed error information, helping clients understand the cause and nature of errors on a field-by-field basis.