


Basic Interview Questions like working method

 Basic Interview Questions

Blockchain Interview Questions

 Blockchain Interview Questions


React.js Interview Questions

 React Interview Questions


Golang Interview Questions

 Golang Interview Questions

Angular Interview Questions

 Angular Interview Questions


Javascript / Node.js Interview Questions

 Node.js Interview Questions

Java Interview Questions

 Java Interview Questions

Jack Wang Interview Question History

 Jack Wang's Interview Questions.

NEW Interview Questions

1. What are the differences between cookies and local storage?

Cookies and local storage are two different ways to store data on the client-side (i.e., in the user's browser) in web development.

Here are some of the key differences between cookies and local storage:

1. **Size Limit:** Cookies have a size limit of 4KB, whereas local storage has a much larger size limit of at least 5MB (depending on the browser).
2. **Expiration:** Cookies can have an expiration date, which means they will be deleted after a specified amount of time. Local storage, on the other hand, does not have an expiration date, and the data will persist until it is manually cleared by the user or by the web application.
3. **Accessibility:** Cookies can be accessed both by the client-side and server-side, while local storage can only be accessed by the client-side.
4. **Security:** Cookies can be set to be HttpOnly, which means they cannot be accessed by JavaScript. Local storage, on the other hand, is accessible by JavaScript by default.
5. **Usage:** Cookies are typically used for sending data back to the server, while local storage is typically used for client-side data storage (such as user preferences, login status, or data caching).

Overall, cookies are better suited for transmitting small amounts of data back and forth between the client and the server, while local storage is better suited for storing larger amounts of data on the client-side for use by the web application.

2. Node.js multi-threading workflow

Node.js is a single-threaded language, meaning it runs code on a single thread of execution. However, it does support multi-threading through the use of worker threads.

Worker threads allow Node.js to create additional threads that can execute JavaScript code independently of the main thread. This can be useful for executing CPU-intensive tasks in parallel, such as image processing or data compression.

Here's a typical workflow for using worker threads in Node.js:

1. Create a worker thread: To create a worker thread, you first need to require the 'worker_threads' module. You can then create a new worker thread by calling the 'Worker' constructor function and passing in the name of the script file that will run in the new thread.
2. Pass data to the worker thread: Once you have created a worker thread, you can pass data to it using the 'postMessage' method. This method takes a single argument, which can be any JavaScript value that can be serialized.
3. Listen for messages from the worker thread: To receive messages from the worker thread, you need to listen for the 'message' event on the worker object. When a message is received, the event handler function will be called, and you can access the message data through the 'data' property of the event object.
4. Send messages back to the main thread: To send messages back to the main thread from the worker thread, you can use the 'parentPort.postMessage' method. This method works in the same way as the 'postMessage' method on the worker object.
5. Terminate the worker thread: When you are finished with a worker thread, you should call the 'worker.terminate' method to terminate the thread and free up system resources.

Overall, the use of worker threads in Node.js allows for more efficient and parallel processing of tasks, which can improve the performance and scalability of your applications.

3. Node.js event loop

Node.js uses an event-driven architecture, with an event loop that handles all incoming events and requests. The event loop is responsible for processing all I/O operations in a non-blocking way, which allows Node.js to handle large numbers of concurrent connections without becoming blocked or unresponsive.

Here's how the event loop works in Node.js:

1. Event loop basics: The event loop is a continuously running loop that waits for events to occur. When an event occurs, the event loop processes it by executing the corresponding callback function.
2. Event queue: All events in Node.js are processed through an event queue. When an event occurs, it is added to the end of the event queue.
3. Call stack: The event loop maintains a call stack, which is a data structure that stores the currently executing function. When an event is added to the event queue, the corresponding callback function is added to the call stack.
4. Non-blocking I/O: Node.js uses non-blocking I/O, which means that I/O operations such as file I/O or network I/O do not block the event loop. Instead, these operations are performed asynchronously, and a callback function is executed when the operation completes.

5. Event loop phases: The event loop in Node.js consists of several phases, each of which is responsible for processing a specific type of event. These phases include the timers phase, I/O callbacks phase, idle, prepare, and close callbacks phase.
6. Microtasks queue: Microtasks are small, lightweight tasks that can be executed immediately after the current event loop phase completes. Promises, `process.nextTick()` and `setImmediate()` functions are examples of microtasks in Node.js.

Overall, the event loop is a critical component of Node.js architecture, as it allows Node.js to handle large numbers of concurrent connections in a non-blocking way, which is essential for building scalable and high-performance applications.

4. Web Security techniques
5. How to configure webpack, babel and vite?
6. DevOps pipeline
7. Why we use react hook(advantage) ?

React Hooks provide a number of advantages over traditional class components in React, which is why they are becoming increasingly popular among React developers. Here are some reasons why you should use React Hooks:

1. Simplifies code: With React Hooks, it's possible to write functional components that have state and other React features, which simplifies the code and makes it easier to read and maintain.
2. Reusability: Hooks are reusable, which means that they can be shared between components, making it easier to encapsulate and reuse logic across your application.
3. Better performance: Hooks can help improve performance by reducing the number of class components needed in your application, which can help reduce the amount of memory and CPU usage.
4. Reduces complexity: By using hooks, you can reduce the complexity of your components and make them more modular, which makes it easier to debug and test your code.
5. Easier to learn: Since hooks are based on functional programming concepts, they can be easier to learn and understand than class components, which can help developers become more productive more quickly.
6. Better interoperability: Hooks can be used with other libraries and tools, which makes it easier to integrate React with other parts of your application or with external APIs.
7. Future-proof: React Hooks have been adopted as part of the React core API, and are likely to become the preferred way to write React components in the future.

Overall, React Hooks provide a more functional programming style to React, which simplifies the code and makes it more modular and reusable. This helps developers to create more scalable and maintainable applications.

8. React context concept and use case.
9. Kubernetes concept, use case and difference with docker.
10. Understanding about terraform
11. Micro-frontend concept and usecase, example
12. Microservice concept, usecase and advantage/disadvantage.
13. AWS amplify, lambda
14. Use case of array and map
15. What motivates you?
16. Tell me about search algorithms and complexity.
17. What is the indexed key in RDS.
18. What is your weakness?
19. What is the difference between AWS Lambda and AWS EC2?

AWS Lambda and AWS EC2 are both AWS services that allow you to run code in the cloud, but they have different functionalities and are designed for different purposes.

AWS Lambda is a serverless compute service that allows you to run code in response to events, such as HTTP requests, changes to data in an S3 bucket, or messages from an Amazon SNS topic.

AWS EC2, on the other hand, is a web service that provides resizable compute capacity in the cloud. With EC2, you can launch and manage virtual machines, called instances, that run various operating systems and applications.

Functionality: Lambda is designed for running small, stateless functions that can be triggered by events, while EC2 is designed for running long-running applications or workloads that require persistent storage, specialized hardware, or custom configurations.

Scalability: Lambda scales automatically to handle any number of requests, while EC2 requires you to manually manage and scale the underlying infrastructure.

Pricing: Lambda charges you based on the number of requests and the compute time used, while EC2 charges you based on the instance type, the usage time, and the data transfer.

Management: Lambda is a fully managed service, which means that AWS takes care of the underlying infrastructure, security, and scaling, while EC2 requires you to manage and maintain the instances yourself.

20. What is microservice and communication between microservices?

Microservices is an architectural style that structures an application as a collection of small, independent services, each of which has a specific business capability and communicates with other services using well-defined APIs.

Synchronous communication: In synchronous communication, the client sends a request to the server and waits for a response before continuing. This type of communication is common in RESTful APIs and RPCs. Synchronous communication can be useful when a response is needed immediately or when the client needs to know the status of a long-running operation.

Asynchronous communication: In asynchronous communication, the client sends a request to the server and does not wait for a response. Instead, the server sends a response later, typically using a message queue or event-driven architecture. Asynchronous communication can be useful when a response is not needed immediately or when the client needs to perform other tasks while waiting for a response.

21. What is the AWS CI/CD and how to implement that?

AWS CI/CD (Continuous Integration/Continuous Delivery) is a set of practices and tools for automating the building, testing, and deployment of software applications on Amazon Web Services (AWS). CI/CD enables developers to quickly and reliably deliver code changes to users.

- * Set up your Elastic Beanstalk environment: Create an Elastic Beanstalk environment to deploy your application to. This environment will serve as the deployment target for your CI/CD pipeline.

- * Create an AWS CodeCommit repository: Create a CodeCommit repository to store your application code.

- * Create an AWS CodeBuild project: Create a CodeBuild project that compiles your application code, creates a deployable artifact, and runs tests.

- * Create an AWS CodePipeline pipeline: Create a CodePipeline pipeline that orchestrates the deployment of your application. This pipeline should have a source stage that pulls code from your CodeCommit repository, a build stage that builds your application using

CodeBuild, and a deploy stage that deploys your application to your Elastic Beanstalk environment.

* Test your pipeline: Test your pipeline by making a change to your application code and verifying that the change is automatically built and deployed to your Elastic Beanstalk environment.

22. What is the pipeline in AWS EC2 and AWS S3 and how to implement that?

In AWS EC2, a pipeline typically refers to a CI/CD pipeline that automates the process of building, testing, and deploying software applications on EC2 instances. This is similar to the process described in the previous question for AWS Elastic Beanstalk, but with EC2 instances as the deployment targets. To implement a pipeline on EC2, you would typically use services such as AWS CodePipeline, AWS CodeBuild, and AWS CodeDeploy, along with other AWS services that are relevant to your specific use case.

In AWS S3, a pipeline typically refers to a data processing pipeline that processes objects stored in S3 buckets. This can involve using AWS services such as AWS Lambda, AWS Glue, or Amazon EMR to perform various operations on S3 objects, such as transforming data, analyzing logs, or performing machine learning tasks. To implement a pipeline on S3, you would typically create a series of AWS services that work together to process data objects in a specific way.

23. What is the Route 53 and Cloudfront in AWS and what is the difference between them?

AWS Route 53 is a highly available and scalable DNS service that allows you to manage your domain names and route traffic to your web applications running on AWS or elsewhere.

AWS CloudFront, on the other hand, is a content delivery network (CDN) service that accelerates the delivery of your web content, such as static and dynamic web pages, images, videos, and streaming media, to users around the world.

Functionality: Route 53 is a DNS service that routes traffic to various endpoints, while CloudFront is a CDN service that accelerates the delivery of web content to users.

Cost: Route 53 charges based on the number of hosted zones, the number of DNS queries, and the use of advanced features, while CloudFront charges based on the volume of data transferred and the number of requests.

Caching: Route 53 does not cache your web content, while CloudFront caches your content at edge locations to improve performance and reduce latency.

Security: Route 53 does not provide security features for web content, while CloudFront provides features such as signed URLs and cookies to protect your content from unauthorized access.

- 24. Which types is there in Postgres database?
- 25. What is react hook and context?
- 26. What is react life cycle and useEffect is componentDidMount?
- 27. What is AWS and how can you deploy web application?
- 28. What is nodejs and which frameworks did you use?
- 29. What is Nginx?