If you're interviewing for a Node.js developer role, it's important to be prepared for advanced questions that can test your expertise and experience.

In this article, Nikita Shevtsiv, Senior Software Engineer and Certified Technical Interviewer at EPAM Anywhere, has compiled 15 advanced Node.js interview questions and answers that can help you prepare for your next [job interview](https://anywhere.epam.com/en/blog/epam-anywhere-job-interviews-what-to-expect). From questions about Node.js modules and event-driven architecture to debugging techniques and performance optimization, these questions cover a range of topics that can help you showcase your Node.js skills and impress your potential employers. Let’s get into it!

1. How do you decide whether to use Node.js Express or Node.js Nest based on the project size?

For smaller projects that require less structure and simplicity, Express can be a suitable choice. Express is a lightweight framework that is easy to set up and allows for flexibility in how you structure your application. It is ideal for building small to medium-sized applications where speed and simplicity are a priority.

On the other hand, for larger projects that require more organization and structure, Nest can be a better choice. Nest is a TypeScript-based framework that is built on top of Express, and it provides a more opinionated and structured approach to building applications. It includes features like dependency injection, modules, and decorators, which make it easier to manage larger and more complex applications.

In summary, if you're working on a smaller project that requires simplicity and speed, you might choose to use Express. However, if you're working on a larger project that requires more structure and organization, Nest might be a better choice. Ultimately, the decision between the two frameworks depends on the specific requirements and goals of your project.

**test yourself in a tech interview with us**

Ready for a real-life test of your skills? Send us your CV, pass prescreening, and get invited to a tech interview with our top experts at EPAM Anywhere.

[yes, find me a job](https://anywhere.epam.com/en/find-me-a-job)

2. Explain 0auth2.0 and its benefits

OAuth2.0 is an authorization framework that allows a user to grant a third party application access to their resources without sharing their credentials (i.e., username and password). It is commonly used to enable users to authenticate and authorize applications to access their data and services without sharing their login credentials.

Here are some of the benefits of using OAuth2.0:

* **Improved security:** OAuth2.0 provides an additional layer of security by eliminating the need for users to share their passwords with third-party applications. Instead, applications are authorized to access the user's resources with a token, which can be revoked by the user at any time.
* **Better user experience:** OAuth2.0 eliminates the need for users to create separate accounts for each application they use. Instead, users can log in to an application using their existing credentials from a trusted provider (e.g., Google, Facebook).
* **Access control:** OAuth2.0 allows users to control the level of access granted to each application. Users can choose to grant read-only access, read-write access, or full access to their resources.
* **Scalability:** OAuth2.0 is widely adopted and supported by many major providers, making it easy for developers to integrate with multiple services and platforms.

Overall, OAuth2.0 provides a secure and convenient way for users to share their resources with third-party applications, without compromising their credentials or data.

3. Compare service-oriented architecture (SOA) and microservice architecture (MSA)

Service-oriented architecture (SOA) and microservice architecture (MSA) are two popular software architecture styles used for building complex software systems. While both of them are used for designing distributed applications, there are some fundamental differences between the two:

* **Architecture style:** SOA is an architecture style that focuses on the decomposition of an application into loosely coupled services, whereas MSA is an architecture style that decomposes an application into independent, small and modular services.
* **Service granularity:** SOA typically deals with larger, coarse-grained services that handle multiple functions, while MSA services are fine-grained and focused on performing a single function.
* **Communication mechanism:** SOA relies heavily on web services and middleware for communication between services, while MSA primarily relies on lightweight communication protocols such as REST or messaging for inter-service communication.
* **Deployment:** SOA services are typically deployed in a centralized manner, with a few large services running on dedicated servers, while MSA services are deployed in a decentralized manner, with small services running on their own servers or containers.
* **Development:** SOA services are developed using a top-down approach, with a focus on standardization and reuse, while MSA services are developed using a bottom-up approach, with a focus on agility and flexibility.
* **Scalability:** SOA services are less scalable than MSA services due to their larger size and centralized deployment, while MSA services are highly scalable due to their small size and decentralized deployment.

In summary, while both SOA and MSA are designed to build complex software systems, they differ in their architecture style, service granularity, communication mechanism, deployment, development approach, and scalability. SOA services are larger and more tightly coupled, while MSA services are smaller and more loosely coupled, making them easier to develop, deploy, and scale.

4. What does “low in coupling and high in cohesion” mean in backend development?

In backend development, "low in coupling and high in cohesion" refers to the design principles that aim to make software systems more modular, maintainable, and scalable.

Low coupling means that the components or modules of the system are designed to be independent of each other. In other words, they have minimal or no dependencies on each other. This design principle makes the codebase easier to maintain, test, and modify because changes in one module do not affect other modules in the system. It also allows for better scalability as new features can be added without disturbing the existing system.

High cohesion, on the other hand, refers to the design principle of grouping related functionality together within a module or component. This means that the functions or methods within a module have a common purpose and work together to achieve that purpose. This design principle also makes the codebase easier to maintain, test, and modify, as changes to one function or method affect only the related functionality.

In summary, low coupling and high cohesion are two important design principles that promote modularity, maintainability, and scalability in software systems. By minimizing dependencies and grouping related functionality, developers can create software systems that are easier to work with and modify, leading to a more stable and maintainable codebase over time.

**related:**

[top 18 backend developer interview questions answered**read more**](https://anywhere.epam.com/en/blog/top-18-backend-developer-interview-questions-answered)

5. How do you ensure the security of backend systems?

There are several measures one can take to ensure the security of backend systems, including:

* **Secure coding practices:** Ensuring that the code written for backend systems is secure is one of the most important ways to ensure system security. Developers should follow secure coding practices such as input validation, output encoding, and avoiding the use of vulnerable libraries.
* **Authentication and access control:** Backend systems should implement strong authentication mechanisms such as multi-factor authentication and role-based access control. This ensures that only authorized personnel have access to the system.
* **Encryption:** Backend systems should encrypt sensitive data at rest and in transit. This includes using secure communication protocols such as HTTPS and SSL/TLS for network communication.
* **Regular security testing:** Regular security testing, including penetration testing and vulnerability assessments, can help identify and address security issues in backend systems.
* **Monitoring and logging:** Backend systems should be continuously monitored for suspicious activity, and logs should be kept for audit and analysis purposes. This can help identify security breaches and provide insight into system weaknesses.
* **Regular updates and patches:** Backend systems should be regularly updated with the latest security patches and updates to ensure that any known vulnerabilities are addressed.

Overall, ensuring the security of backend systems requires a holistic approach that includes secure coding practices, strong authentication and access control, encryption, regular security testing, monitoring and logging, and regular updates and patches.

6. What’s the difference between using PostgreSQL and MongoDB for Node.js server projects?

PostgreSQL and MongoDB are both popular [databases](https://anywhere.epam.com/en/blog/dns-as-your-edge-database) used for Node.js server projects, but they have some key differences that can affect their suitability for specific use cases. Here are some of the main differences between the two:

* **Data structure:** PostgreSQL is a relational database, which means it stores data in tables with predefined relationships between them. On the other hand, MongoDB is a NoSQL database that stores data as JSON-like documents.
* **Scalability:** MongoDB is designed for scalability and can handle large amounts of unstructured data, making it a good choice for big data and real-time applications. PostgreSQL, on the other hand, may require more setup and optimization to handle large data sets.
* **Query language:** PostgreSQL uses SQL, a standardized query language, while MongoDB uses its own query language, which is based on [JavaScript](https://anywhere.epam.com/en/blog/javascript-interview-questions-at-epam-anywhere). If you're familiar with SQL, PostgreSQL may be easier to work with, but if you're comfortable with JavaScript, you may prefer MongoDB.
* **ACID compliance:** PostgreSQL is fully ACID-compliant, meaning it ensures data consistency and accuracy even in the event of a system failure. MongoDB, on the other hand, sacrifices some of the guarantees of ACID compliance in favor of performance and scalability.
* **Community and support:** Both PostgreSQL and MongoDB have large and active communities, but PostgreSQL has been around longer and has a more established support network.

In summary, the choice between PostgreSQL and MongoDB largely depends on the specific needs of your Node.js project. If you need a database that can handle large amounts of unstructured data and prioritize scalability, MongoDB might be a better choice. On the other hand, if you require a fully ACID-compliant database with a proven track record, PostgreSQL may be a better fit.

**related:**

[Node JS resume example**read more**](https://anywhere.epam.com/en/blog/node-js-developer-resume-example)

7. How do you decide when to implement caching in your backend systems?

Caching is a technique that can help improve the performance and scalability of backend systems by storing frequently accessed data in memory or on disk, allowing it to be retrieved more quickly. However, it's important to carefully consider when and where to implement caching, as it can also introduce complexity and potential tradeoffs.

Here are some factors to consider when deciding whether to implement caching in your backend systems:

* **Frequency of data access:** Caching is most effective when data is frequently accessed and rarely updated. If a piece of data is only accessed once in a while, caching it may not provide a significant performance improvement.
* **Data size and complexity:** Caching large or complex data structures can be expensive in terms of memory and processing overhead. It's important to balance the benefits of caching with the cost of storing and retrieving the cached data.
* **Latency and response time requirements:** If your system requires very low latency and fast response times, caching can be an effective way to achieve those goals. However, if your system can tolerate some delay or variability in response times, caching may not be as important.
* **Scalability requirements:** Caching can help improve the scalability of a system by reducing the load on backend databases or other services. If your system needs to handle high volumes of traffic, caching can be a valuable tool for managing that load.
* **Data consistency requirements:** Caching introduces the potential for data inconsistency, since the cached data may not always be up to date with the latest changes. It's important to carefully consider the impact of stale or outdated data on your system and whether the benefits of caching outweigh that risk.

Overall, the decision to implement caching in your backend systems should be based on a careful evaluation of your specific requirements and constraints, taking into account factors such as data access patterns, latency and response time requirements, scalability, and data consistency.

8. How does the architecture of a Nest.js application differ from other Node.js frameworks like Express.js?

Nest.js is a popular Node.js framework that uses modern JavaScript and TypeScript features to build scalable and efficient applications. The architecture of a Nest.js application is based on the principles of the Model-View-Controller (MVC) [design pattern](https://anywhere.epam.com/en/blog/less-popular-javascript-design-patterns) and draws inspiration from Angular's architecture. Here's an overview of the architecture of a Nest.js application:

* **Modules:** The application is divided into multiple modules, each responsible for a specific set of features. Each module contains its own set of controllers, services, and providers.
* **Controllers:** Controllers are responsible for handling incoming HTTP requests and returning the appropriate HTTP responses. Controllers are linked to a specific route and can have multiple endpoints.
* **Services:** Services are responsible for performing business logic and interacting with the data layer. Services can be shared across multiple modules and can have dependencies injected using the built-in dependency injection system.
* **Providers:** Providers are a type of service that can be instantiated multiple times within a module. Providers can be used to create custom classes, factories, or utilities that can be used across the application.
* **Middleware:** Nest.js supports middleware, which are functions that are executed before or after a request is handled by a controller. Middleware can be used to perform operations like logging, authentication, or error handling.

Compared to other Node.js frameworks like Express.js, Nest.js provides a more structured approach to building applications. Nest.js encourages the use of [TypeScript](https://anywhere.epam.com/en/blog/typescript-vs-javascript-a-chief-software-engineer-s-perspective" \t "_self) and uses decorators to define controllers, services, and providers, which can help improve code readability and maintainability.

Nest.js also includes built-in support for features like dependency injection, middleware, and WebSocket support, which can help simplify application development. Additionally, Nest.js integrates well with other popular Node.js libraries like TypeORM and GraphQL, making it a popular choice for building scalable web applications.

9. How can you test Nest.js applications?

Nest.js applications can be tested using a variety of approaches. Here are some steps to consider:

* **Unit tests:** Write unit tests for each module, service, controller, and other components of your application using a testing framework such as Jest. Ensure that each unit test runs independently of other tests and provides a clear, concise output of the results.
* **Integration tests:** Test the interaction between the different modules and components of your application using integration tests. This will help you to identify any issues that may arise due to dependencies between different parts of your code.
* **End-to-end (e2e) tests:** Use e2e testing frameworks such as Cypress or Protractor to test your application from the user's perspective. These tests simulate the user's actions and interactions with your application, and can help you to identify issues related to user experience and usability.
* **Mocking:** Use mocking frameworks such as Sinon or Jest to simulate dependencies in your application. This will help you to isolate and test individual components without having to rely on external resources or services.
* **Code coverage:** Use a code coverage tool such as Istanbul to ensure that your tests cover as much of your code as possible. This will help you to identify any areas of your code that are not being tested adequately.
* **Continuous integration (CI):** Use a CI tool such as Travis CI or CircleCI to automate your testing process. This will help you to catch any issues early in the development process and ensure that your code is always in a deployable state.

By following these steps, you can ensure that your Nest.js application is thoroughly tested and ready for production.

10. What is an interceptor in Nest.js?

In Nest.js, an interceptor is a middleware that can intercept incoming requests and outgoing responses. Interceptors provide a way to modify the request or response objects, execute additional logic, or even terminate the request/response cycle prematurely.

Interceptors can be used to implement cross-cutting concerns, such as authentication, logging, error handling, and caching. They are registered globally, per module, or per controller and can be synchronous or asynchronous.

11. Can you describe the role of modules in a Nest.js project?

In Nest.js, modules are a fundamental building block of the application architecture. Modules help to organize and structure the codebase into smaller, more manageable units, which can be developed, tested, and deployed independently.

A module can be seen as a container for a specific feature or domain of the application. Each module can have its own controllers, services, providers, and other related components, which are encapsulated within the module's scope.

Modules can also be used to define the dependencies and relationships between different parts of the application. For example, a module can import other modules to access their functionality or export its own functionality for use by other modules.

Some of the key roles of modules in Nest.js include:

* **Encapsulation:** Modules encapsulate related functionality, making it easier to reason about and maintain the codebase.
* **Dependency management:** Modules define the dependencies of a feature or domain, ensuring that the necessary components are available and properly configured.
* **Reusability:** Modules can be reused in other applications or projects, providing a modular and reusable architecture.
* **Testability:** Modules can be tested in isolation, allowing for easier and more thorough testing of individual features or domains.

Overall, modules provide a powerful mechanism for structuring and organizing the codebase in a modular and maintainable way, which is essential for developing robust and scalable applications.

12. How would you describe a good approach to deploying a backend project?

Here is a step-by-step approach to backend project deployment:

1. **Prepare your application for deployment:** Before deploying your application, you need to ensure that it is properly configured and optimized for production use. This includes setting environment variables, configuring any necessary database connections, and optimizing any performance-related settings.
2. **Choose a hosting provider:** There are many hosting providers available that can host your Express application, such as [AWS](https://anywhere.epam.com/en/blog/what-is-aws), Google Cloud Platform, Heroku, and Digital Ocean. Consider your budget, deployment needs, and other factors when selecting a provider.
3. **Create a deployment environment:** Once you have chosen a hosting provider, create a deployment environment that mirrors your production environment. This typically involves setting up a virtual machine or container with the necessary dependencies and configurations.
4. **Install dependencies:** Install any necessary dependencies using a package manager like npm. Ensure that you only include production dependencies, not development dependencies.
5. **Build and bundle your application:** Use a build tool like webpack or gulp to bundle your application's code and assets into a single file or set of files. This can help optimize performance and reduce load times.
6. **Test your application:** Before deploying your application, test it in your deployment environment to ensure that everything is working as expected. This may include functional testing, performance testing, or security testing.
7. **Deploy your application:** Once you are confident that your application is ready to be deployed, use a deployment tool like Git, FTP, or a CI/CD pipeline to deploy your application to your hosting provider.
8. **Monitor and maintain your application:** After deployment, monitor your application to ensure that it is running smoothly and perform regular maintenance tasks, such as updating dependencies and performing security updates.

**related:**

[top 23 senior software engineer interview questions**read more**](https://anywhere.epam.com/en/blog/senior-software-engineer-interview-questions)

13. Can you give an example of a project that required WebSocket communication?

Imagine you're building a real-time multiplayer game where players can move around a virtual world and interact with each other. To make this work, you need to establish a persistent, bidirectional communication channel between the game client (running in the player's web browser) and the game server (running on a remote server).

One way to achieve this is by using WebSockets. With WebSockets, the client and server can exchange data in real time, without the need for repeated HTTP requests/responses. The server can push updates to the client whenever something important happens (e.g., a player moves or scores points), and the client can send messages to the server whenever the player takes an action (e.g., moves, attacks, chats).

To implement this, you might use a WebSocket library like socket.io (for Node.js), which abstract away some of the lower-level details of WebSocket communication. You would need to create a WebSocket server on the backend, which listens for incoming WebSocket connections and handles incoming/outgoing messages. You would also need to create a WebSocket client on the frontend, which establishes a connection to the server and sends/receives messages as needed.

With this infrastructure in place, you could then implement the game logic on top of the WebSocket communication layer. For example, when a player moves, the client would send a "move" message to the server over the WebSocket connection. The server would receive this message, update the game state accordingly, and then broadcast the new state to all connected clients. Each client would receive the updated state and redraw the game world to reflect the changes. This would happen in real time, with no need for page refreshes or long polling.

[](https://info.anywhere.epam.com/cs/c/?cta_guid=8d3e1f36-f249-47c1-989b-511e6b19a257&signature=AAH58kH-oRtqMQUXt6S4ovE_VVZYbo15Bw&placement_guid=37f0456c-8f2c-4ce5-9058-d169d0954804&click=2ffc98b0-c868-4e6e-9954-1d760c2d0c39&hsutk=ff4dd194e2ff06fe9af7817a37ae8f91&canon=https%3A%2F%2Fanywhere.epam.com%2Fen%2Fblog%2Fadvanced-node-js-interview-questions-answers&portal_id=20543775&redirect_url=APefjpEk9bYAgQnbXca-6_gC5ZYjSmvftpM7pNZhmojBv_qrBtSWiRSMMGUpMwbcRs4cA1rDb3PA4vCWcFaI0iPXYqCzM8EGHriR4MbCiQ-8BHZ3mUJI1evUVK4J0cWnG3cSpoGcRS7P9O2VuJdTo5lCPPS4pIrdBDhVwSjjE8acXFDfRQGg-XU&__hstc=249402712.ff4dd194e2ff06fe9af7817a37ae8f91.1699193084808.1699193084808.1699193084808.1&__hssc=249402712.1.1699193084808&__hsfp=3513193844)

14. List pros and cons of using GraphQL vs REST API approaches

[GraphQL](https://anywhere.epam.com/en/blog/tutorial-introduction-to-graphql-using-apollo-in-javascript) and [REST](https://anywhere.epam.com/en/blog/how-to-secure-rest-api-spring-boot-okta) are both widely used approaches for building APIs.

**Pros of GraphQL:**

* **Flexible data querying:** With GraphQL, clients can specify exactly what data they need, and the server will only return that data. This reduces the amount of over-fetching or under-fetching of data and makes it easier to manage complex queries.
* **Efficient data loading:** GraphQL allows you to fetch multiple resources in a single query, which can improve performance and reduce network overhead.
* **Versioning is easier:** With REST APIs, versioning is often a challenge. However, GraphQL's schema-based approach makes versioning easier since changes to the schema can be made without breaking existing clients.
* **Self-documenting:** GraphQL APIs are self-documenting, meaning that clients can easily explore the API schema and understand how to use it.

**Cons of GraphQL:**

* **Increased complexity:** GraphQL requires a more complex infrastructure, and it can be more difficult to implement than REST.
* **Caching can be difficult:** Since GraphQL queries are often more dynamic than REST queries, caching can be more difficult to implement.
* **Learning curve:** The learning curve for GraphQL can be steep, especially for developers who are used to working with REST APIs.

**Pros of REST:**

* **Simplicity:** REST APIs are easy to understand and implement. They are a well-established standard and have a large community of developers working with them.
* **Caching is straightforward:** REST APIs are designed to be cacheable, making it easy to implement caching strategies to improve performance.
* **No learning curve:** REST APIs are simple and familiar to many developers, so there is no significant learning curve to get started.

**Cons of REST:**

* **Over-fetching or under-fetching of data:** With REST APIs, clients often receive more data than they need, or not enough. This can lead to performance issues and wasted network resources.
* **Versioning can be difficult:** With REST APIs, versioning can be challenging since changes to the API can break existing clients.
* **Multiple requests required:** With REST APIs, clients often need to make multiple requests to fetch all the data they need. This can increase latency and network overhead.

In summary, GraphQL provides more flexibility and efficient data loading but comes with increased complexity and a steeper learning curve. REST APIs are simpler and easier to implement but can be less flexible and require more requests to fetch all the data needed. The choice between GraphQL and REST will depend on the specific needs of your project and the resources available to you.