Exam preparation materials

1. Node.js architecture Node.js 架构

Node or Node.js is a software platform based on the V8 engine (which translates JavaScript into machine code) that transforms JavaScript from a highly specialized language into a general-purpose language.

Node.js adds the ability for JavaScript to：

1. interact with I / O devices through its API (written in C ++),

2. connect other external libraries written in different languages,

3. providing calls to them from JavaScript code.

Node.js is a cross-platform runtime environment and library for running JavaScript applications outside the browser. It is used for creating server-side and networking web applications. It is open source and free to use.

Node.js can be used to build different types of applications such as command line application, web application, real-time chat application, REST API server etc. However, it is mainly used to build network programs like web servers, similar to PHP, Java, or ASP.NET.

Node.js is used mainly on the server, acting as a web server, but it is possible to develop on Node.js and desktop window applications and even program microcontrollers. At the core of Node.js is event-driven and asynchronous (or reactive) programming with non-blocking I / O.

Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms.

All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request.

So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

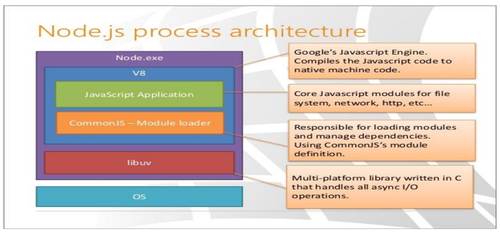
An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes.

Internally, Node.js uses libev for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

Node.js implements CommonJS modules standard. CommonJS is a group of volunteers who define JavaScript standards for web server, desktop, and console application.

The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application.


Node.js process architecture
Node.exe
V8
JavaScript Application
CommonJS – Module loader
libuv
OS
Google's Javascript Engine. Compiles the Javascript code to native machine code.
Core Javascript modules for file system, network, http, etc...
Responsible for loading modules and manage dependencies. Using CommonJS's module definition.
Multi-platform library written in C that handles all async I/O operations.

2. Node.js package management Node.js 包管理

NPM (Node Package Manager) is a package management tool for Node.js, used to install, manage, and share JavaScript packages and modules. It is a core component of the Node.js ecosystem. Node Package Manager provides two main functionalities:

It provides online repositories for node.js packages/modules which are searchable on search.nodejs.org

It also provides command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

NPM performs the operation in two modes: global and local. In the global mode, NPM performs operations which affect all the Node.js applications on the computer whereas in the local mode, NPM performs operations for the particular local directory which affects an application in that directory only.

Here are some key points about NPM:

Package Management: NPM provides a command-line interface that allows developers to install, update, uninstall, and manage dependencies of packages in their projects.

Packages and Modules: NPM hosts a vast collection of packages and modules that can be used for various purposes, such as front-end development, back-end development, build tools, testing frameworks, and more. Developers can use NPM to bring these packages and modules into their projects for use in their own code.

package.json: Every Node.js project includes a file called package.json, which defines the metadata and dependencies of the project. The package.json contains information such as the project's name, version, author, license, and lists the other packages and modules that the project depends on.

Version Management: NPM uses Semantic Versioning to manage package versions. By specifying version ranges for dependencies in the package.json, it ensures that the project's dependency relationships are handled correctly and allows for automated updates.

Global Installation and Local Installation: NPM provides both global and local installation options. Global installation installs packages into the system's global directory, making them available for use in any project. Local installation installs packages into the local directory of the project, making them only accessible within that project.

NPM Scripts: NPM allows developers to define custom script commands in the package.json. These scripts can be used to perform various tasks, such as building, testing, running development servers, and more.

NPM Registry: The default registry for NPM is npmjs.com, where developers can publish and share their packages. Additionally, there are other NPM registries available, including private registries for internal company use.

In summary, NPM is a package management tool for Node.js that enables the easy installation, management, and sharing of JavaScript packages and modules.

3. Node.js CLI

The Node.js CLI (Command Line Interface) is a command-line tool that allows developers to interact with Node.js and perform various tasks related to Node.js development. It provides a set of commands that can be executed in a terminal or command prompt to perform actions such as running scripts, managing dependencies, and executing Node.js programs. Here are some commonly used commands of the Node.js CLI:

node: The node command is used to execute a JavaScript file with Node.js. For example, running node script.js will execute the script.js file using the Node.js runtime.

npm: The npm command is used to interact with the Node Package Manager (NPM). It allows you to install packages, manage dependencies, run scripts defined in the package.json file, and perform various other package-related tasks. For example, npm install installs the dependencies specified in the package.json.

npx: The npx command is used to execute Node.js packages without the need for global installation. It allows you to run a package directly from the command line without installing it globally. For example, npx webpack will run the webpack package without the need to install it globally.

node --version: This command returns the version of Node.js installed on your system.

npm --version: This command returns the version of NPM installed on your system.

npm init: This command initializes a new Node.js project by creating a package.json file. It prompts you to enter details such as project name, version, description, and entry point.

npm install <package-name>: This command installs a specific package from the NPM registry and adds it as a dependency in the package.json file.

npm run <script-name>: This command runs a script defined in the scripts section of the package.json file. It is commonly used to execute build scripts, start development servers, or run tests.

4. Node.js packages

Node.js has a vast ecosystem of packages available through the npm (Node Package Manager) registry. These packages provide ready-to-use functionality, libraries, and tools that you can include in your Node.js projects. Here are some popular Node.js packages across different categories:

Express.js

Express.js is a fast, robust and asynchronous Model-View-Controller framework for Node.js. It helps to direct server and routes. It helps to design various web applications and based on passing arguments to templates. It allows to dynamically render HTML Pages.

Nest.js

Nest.js is an adaptable and versatile REST API framework for building efficient and scalable Node.js server-side applications. It is shaped with TypeScript that helps to maintain compatibility with pure JavaScript and integrates modules of Object-Oriented Programming, Functional Programming, and Functional Reactive Programming.

Hapi.js

Hapi is an open-source, stable and constant Model-View-Controller MVC framework for structuring web applications and services. Hapi.js provides an effortless structuring of API (application programming interface) servers, websites, and HTTP(Hypertext Transfer Protocol) proxy applications. Because of its robust plugin system.

It enables you to add new features and fix bugs at a swift pace. Hapi.js provides you with the features of routing, input, output validation, and caching that assists in structuring REST APIs. It’s easy to build an API that serves clients’ needs for mobile and single-page applications.

Koa.JS

Koa.js is a Powerful HTTP middleware framework for node.js to create web applications and APIs more fun to draft. Its middleware stack runs in a stack-like method, enabling you to implement downstream then refine and manage the response upstream. This incorporates elements like content negotiation, normalization of node discrepancies, redirection, and more.

package.json

The "package.json" file is used to hold the metadata about a particular project. This information provides the Node package manager the necessary information to understand how the project should be handled along with its dependencies.The package.json files contain information such as the project description, the version of the project in a particular distribution, license information, and configuration data.The package.json file is normally located at the root directory of a Node.js project.

5. Node.js app (project folder structure)

The folder structure of a project can vary depending on the specific requirements, programming language, framework, and personal/team preferences. However, here is a general guideline for organizing a project folder structure:

Root Folder:

1. README.md: Document describing the project and its usage.
2. LICENSE: License file specifying the project's terms of use.
3. .gitignore: File specifying the files and directories to be ignored by version control (e.g., Git).

Source Code:

src/: The main source code directory.

- app/: Directory for the application-specific code.
    - controllers/: Contains the code for handling request/response logic.
    - models/: Contains the data models or business logic.
    - views/: Contains the templates or UI-related code (for web applications).

services/: Contains reusable services or helper functions.

config/: Configuration files for the project (e.g., database configuration, environment variables).

routes/: Contains the route definitions or URL mappings.

utils/: Utility functions or modules.

tests/: Directory for test files.

Build/Compiled Output:

dist/: Directory for compiled or transpiled code (for languages like JavaScript, TypeScript).

bin/: Directory for executable files (for compiled languages like C++, Go).

Resources:

public/: Directory for publicly accessible files (e.g., static assets, client-side JavaScript, CSS).

assets/: Directory for project-specific assets (e.g., images, fonts).

templates/: Directory for template files (for server-side rendering).

Documentation:

docs/: Directory for project documentation (e.g., API documentation, design documents).

reports/: Directory for project reports or analysis.

Configuration:

config/: Configuration files specific to the deployment environment (e.g., server configurations, deployment scripts).

.env: Environment-specific configuration file (e.g., database credentials, API keys).

Dependencies:

node_modules/: Directory for third-party libraries and dependencies (for Node.js projects).

vendor/: Directory for third-party libraries and dependencies (for other programming languages).

Build/Package Management:

package.json: File specifying project metadata, dependencies, and build scripts.

pom.xml: File specifying Maven dependencies (for Java projects).

Other:

logs/: Directory for log files generated by the application.

temp/: Directory for temporary files or cache.

scripts/: Directory for project-related scripts (e.g., build scripts, deployment scripts).

6. Microservice architecture

Microservices architectures, where applications include multiple low coupled components, which can be developed using different languages frameworks, and platforms, have become one of the most popular alternatives to traditional monolithic enterprise applications.

Microservice architecture divides the single application into a small set of services, each running on its own but communicating with each other through APIs.

These services are deployed independently using a fully automated environment. There is an absolute minimum of centralized management of these services.

Since these modules are deployed independently and only communicate over standard protocols or APIs, they lend themselves to continuous delivery and continuous integration processes, reducing costs and risks of developing new features.

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are：

1．Highly maintainable and testable

2．Loosely coupled

3．Independently deployable

4．Organized around business capabilities

5．Owned by a small team

Advantages：

1．Developer independence− Small teams work in parallel and can iterate faster than large teams.

2．Isolation and stability− If a component dies, you spin up another while and the rest of the application continues to function.

3．Technology heterogeneity − Microservice supports different technologies to communicate with each other in one business unit, which helps the developers to use the correct technology at the correct place. By implementing a heterogeneous system, one can obtain maximum security, speed and a scalable system.

4．Small in size − Microservices is an implementation of SOA design pattern. It is recommended to keep your service as much as you can. Basically, a service should not perform more than one business task, hence it will be obviously small in size and easy to maintain than any other monolithic application.

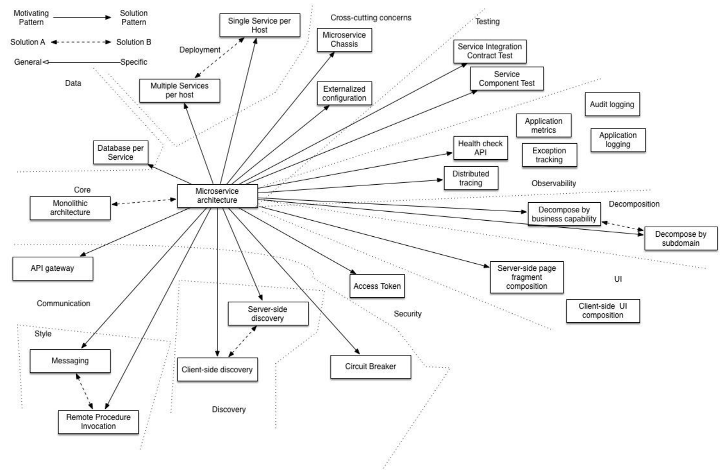
5．Autonomous − Each microservice should be an autonomous business unit of the entire application. Hence, the application becomes more loosely coupled, which helps to reduce the maintenance cost.

6．Lifecycle automation− Individual components are easier to fit into continuous delivery pipelines and complex deployment scenarios not possible with monoliths.

7．Relationship to the business− Microservice architectures are split along business domain boundaries, increasing independence and understanding across the organization.

8．Focused − As mentioned earlier, each microservice is designed to deliver only one business task. While designing a microservice, the architect should be concerned about the one point of the service. By definition, one microservice should be full stack in nature and should be committed to delivering only one business property.

7. Design patterns used in microservice architecture



Discovery Service

In order to make a request, your code needs to know the network location (IP address and port) of a service instance.service instances registered with and unregistered from the service registry： A service instance is responsible for registering itself with the service registry. On startup the service instance registers itself (host and IP address) with the service registry and makes itself available for discovery. The client must typically periodically renew its registration so that the registry knows it is still alive. On shutdown, the service instance unregisters itself from the service registry.

Externalized configuration

enable a service to run in multiple environments without modification： A separate service can be used for centralized storage of service settings in a microservice system. This service should also provide services access to their part of the configuration information. Externalize all application configuration including the database credentials and network location. On startup, a service reads the configuration from an external source, e.g. OS environment variables, etc.

Circuit Breaker

prevent a network or service failure from cascading to other services： The basic idea is to stop cascading failure in a distributed system. A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

The API gateway pattern

The API Gateway is the entry point to all the services that your application is providing. It’s responsible for service discovery (from the client side), routing the requests coming from external callers to the right microservices, and fanning out to different microservices if different capabilities were requested by an external caller.Using an API gateway has the following benefits:

1. Isolates the clients from how the application is partitioned into microservices.
2. Isolates the clients from the problem of determining the locations of service instances
3. Provides the optimal API for each client
4. Reduces the number of requests. For example, the API gateway enables clients to retrieve data from multiple services with a single round-trip. Fewer requests also means less overhead and improves the user experience. An API gateway is essential for mobile applications.
5. Simplifies the client by moving logic for calling multiple services from the client to API gateway
6. Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally

the clients of a Microservices-based application access the individual services:Implement an API gateway that is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services.

Security

A microservice architecture is a distributed architecture. Each external request is handled by the API gateway and at least one service.Each service must implement some aspects of security.In order to implement security in a microservice architecture we need to determine who is responsible for authenticating the user and who is responsible for authorization.Clients authenticate with the API gateway. API clients include credentials in each request.

Login-based clients POST the user’s credentials to the API gateway’s authentication and receive a session token. Once the API gateway has authenticated a request, it invokes one or more services.

JWT

One such popular standard for transparent tokens is the JSON Web Token (JWT). JWT is standard way to securely represent claims, such as user identity and roles, between two parties.

A JWT has a payload, which is a JSON object that contains information about the user, such as their identity and roles, and other metadata, such as an expiration date. It’s signed with a secret that’s only known to the creator of the JWT, such as the API gateway and the recipient of the JWT, such as a service.

First a user has to log in to the authorization service with his/her login data. The authorization service now checks the login data and creates a new JWT.

The name of the user is set as the subject; the claim method sets the user's payload for the authorization information. This is followed by a unique ID by means of which the token can be identified at any time. Finally, an expiry date is set. The token is signed with the private key and serialized into a compact string before it is returned.

Now the user can send the compact token seen in the following listing in the browser's authorization header in order to authenticate him-/herself to the website. The individual parts of the JWT are separated by periods

By design, a service will perform the request operation after verifying the JWT's signature and expiration date.

As a result, there's no practical way to revoke an individual JWT that has fallen into the hands of a hateful third party.

The solution is to issue JWTs with short expiration times, because that limits what a malicious party could do.

One drawback of short-lived JWTs, is that the application must somehow continually reissue JWTs to keep the session active. Fortunately, this is one of the many protocols that are solved by a security standard calling OAuth 2.0

OAuth 2.0

OAuth 2.0 is an authorization protocol that was originally designed to enable a user of a public cloud service, such as GitHub or Google, to grant a third-party application access to its information without revealing its password.

For example, OAuth 2.0 is the mechanism that enables you to securely grant a third party cloud-based Continuous Integration (CI) service access to your GitHub repository

The sequence of events is as follows:

The client makes a request, supplying its credentials using basic authentication.

The API gateway makes an OAuth 2.0 Password Grant request to the OAuth 2.0 authentication server.

The authentication server validates the API client's credentials and returns an access token and a refresh token.

The API gateway includes the access token in the requests it makes to the services. A service validates the access token and uses it to authorize the request.

A service invoked by the API gateway needs to know the principal making the request.

It must also verify that the request has been authenticated. The solution is for the API gateway to include a token in each service request.

The sequence of events for API clients is as follows:

1. A client makes a request containing credentials.

2. The API gateway authenticates the credentials, creates a security token, and passes that to the service or services.

The sequence of events for login-based clients is as follows:

1. A client makes a login request containing credentials.

2. The API gateway returns a security token.

3. The client includes the security token in requests that invoke operations.

4. The API gateway validates the security token and forwards it to the service or services.

Health check API pattern

detect that a running service instance is unable to handle requests:

A service has an health check API endpoint (e.g. HTTP /health) that returns the health of the service. The API endpoint handler performs various checks, such as:

1. the status of the connections to the infrastructure services used by the service instance

2. the status of the host, e.g. disk space

3. application specific logic

A health check client - a monitoring service, service registry or load balancer - periodically invokes the endpoint to check the health of the service instance.

Log aggregation pattern

understand the behavior of users and the application and troubleshoot problems:It is useful to know what actions a user has recently performed: customer support, compliance, security, etc.Record user activity in a database.

8. Authentication in microservice information systems

Authentication in microservice architecture involves the process of verifying the identity of users or services accessing the system's resources. It ensures that only authorized entities can perform certain actions or access sensitive data.

Here are some considerations and patterns for implementing authentication in microservice information systems:

1. Single Sign-On (SSO): SSO enables users to authenticate once and then accessmultiple services or applications without the need to reauthenticate for each one. It can be implemented using protocols such as OAuth 2.0 or OpenID Connect, where a centralized authentication server issues and verifies tokens for authentication.

2. Token-based Authentication: In this approach, when a user or service successfully authenticates, an authentication token (such as a JSON Web Token or JWT) is issued. This token is then included in subsequent requests to authenticate and authorize the user or service. Each microservice can validate the token independently without the need for centralized session management.

3. API Gateway Authentication: The API gateway can handle authentication at a central entry point for all incoming requests. It can perform authentication and authorization checks before routing the request to the appropriate microservice. This pattern simplifies authentication enforcement and consolidates security concerns.

4. Service-to-Service Authentication: Microservices often need to communicate with each other securely. Service-to-service authentication involves using mechanisms such as mutual TLS (Transport Layer Security) or API keys to authenticate and establish secure communication channels between microservices.

5. Role-Based Access Control (RBAC): RBAC is a widely used authorization model that assigns specific roles to users or services and defines their access permissions based on those roles. Microservices can implement RBAC to control access to resources and ensure that only authorized entities can perform certain operations.

6. Centralized Identity and Access Management (IAM): A centralized IAM system can manage user identities, authentication, and authorization across the microservices ecosystem. It provides a unified view of user accounts, roles, and permissions, ensuring consistency and reducing duplication of identity-related logic across services.

7. Secure Token Exchange: In scenarios where one microservice needs to access another microservice on behalf of a user, secure token exchange can be employed. The initial token obtained during user authentication can be exchanged for a limited-use token with restricted permissions. This pattern helps minimize the exposure of sensitive user tokens and enhances security.

8. Multi-factor Authentication (MFA): MFA adds an extra layer of security by requiring users to provide multiple forms of identification to authenticate. It can include factors such as passwords, biometrics, one-time passwords, or hardware tokens. MFA can be implemented at the authentication layer to strengthen the security of microservices.

9. Serverless applications

Serverless was first used to describe applications that significantly or fully incorporate third-party, cloud-hosted applications and services, to manage server-side logic and state.

These are typically “rich client” applications—think single-page web apps, or mobile apps— that use the vast ecosystem of cloud-accessible databases (e.g., Parse, Firebase), authentication services (e.g., Auth0, AWS Cognito), and so on.

Serverless can also mean applications where server-side logic is still written by the application developer, but, unlike traditional architectures, it’s run in stateless compute containers that are event-triggered, ephemeral (may only last for one invocation), and fully managed by a third party.

One way to think of this is “Functions as a Service” or "FaaS". AWS Lambda is one of the most popular implementations of a Functions-as-a-Service platform at present, but there are many others, too.

In the original version, all flow, control, and security was managed by the central server application.

In the Serverless version there is no central arbiter of these concerns. Instead we see a preference for choreography over orchestration, with each component playing a more architecturally aware role—an idea also common in a microservices approach.

10. FaaS services

Fundamentally, FaaS is about running backend code without managing your own server systems or your own long-lived server applications.

FaaS offerings do not require coding to a specific framework or library. FaaS functions are regular applications when it comes to language and environment. For instance, AWS Lambda functions can be implemented “first class” in Javascript, Python, Go, any JVM language (Java, Clojure, Scala, etc.), or any .NET language.

Deployment is very different from traditional systems since we have no server applications to run ourselves. In a FaaS environment we upload the code for our function to the FaaS provider, and the provider does everything else necessary for provisioning resources, instantiating VMs, managing processes, etc.

11. Code review tools

ESLint is a tool for identifying and reporting on patterns found in ECMAScript/JavaScript code, with the goal of making code more consistent and avoiding bugs. ESLint is completely pluggable, every single rule is a plugin and you can add more at runtime.ESLint is a popular JavaScript code analysis tool used for static code checking and identifying potential issues. It helps enforce coding styles, discover common errors and vulnerabilities, and provides a configurable set of rules. ESLint can be easily integrated into Node.js projects and used with various editors and build tools.

JSHint: JSHint is another commonly used JavaScript code analysis tool similar to ESLint. It provides a configurable set of rules to check for potential issues and errors in code. JSHint has fewer configuration options compared to ESLint but is still a good choice for simple code checking tasks.

12. Code formatter

A code formatter, also known as a code beautifier, is a tool that automatically formats source code according to predefined style guidelines. It helps ensure consistent and readable code by applying consistent indentation, spacing, line breaks, and other formatting rules.

It enforces a strict and opinionated code style and can be integrated into various editors and build tools. Prettier focuses on code formatting and aims to minimize stylistic debates among team members.

Prettier is an opinionated code formatter with support for TypeScript, CSS, HTML and so on. It removes all original styling and ensures that all outputted code conforms to a consistent style.

Building and enforcing a style guide： Allows you to create a fully automatic and highly customizable guide style.

Thanks to the automatic change of the code, this approach will allow all project participants to accept it at once.

Helping Newcomers： it’s useful for people with very limited programming experience, and lets quicken the ramp up time from skilled engineers joining the company, as they likely used a different coding style before, and developers coming from a different programming language.

Writing code： With Prettier editor integration, you can just press that magic key binding and poof, the code is formatted

Clean up an existing codebase： It has the ability to convert an already existing codebase according to the new code style