I. UI Development (HTML, CSS, JavaScript, React JS)

A. HTML (HyperText Markup Language)

HTML forms the structural foundation of web pages. While seemingly straightforward, a deep understanding of semantic HTML and accessibility is crucial.

- **Semantic HTML:**
    - **Meaningful Structure:** Instead of relying solely on and , semantic tags like <article>, <nav>, <aside>, <header>, <footer>, <section>, <main> convey the meaning and purpose of the content to both browsers and assistive technologies. This improves SEO, accessibility, and code readability.
    - **Benefits:**
        - **SEO (Search Engine Optimization):** Search engines can better understand the content, leading to improved ranking.
        - **Accessibility:** Screen readers and other assistive technologies can interpret the content more accurately, providing a better user experience for people with disabilities.
        - **Readability and Maintainability:** Code becomes more self-documenting and easier for developers to understand and maintain.
    - **Key Semantic Tags:** Understand the purpose and usage of each structural tag.
- **HTML Forms:**
    - **Advanced Form Elements:** Beyond basic <input> types, explore <select>, <textarea>, <datalist>, <output>, <progress>, <meter>, and their attributes.
    - **Form Validation:** Client-side validation using HTML5 attributes (required, pattern, min, max, type) and JavaScript for more complex scenarios. Understand the importance of both client-side and server-side validation.
    - **Accessibility in Forms:** Using <label> with for attribute, providing clear instructions, handling errors effectively for screen readers.
- **Accessibility (a11y):**
    - **WAI-ARIA (Web Accessibility Initiative - Accessible Rich Internet Applications):** Understand ARIA roles, states, and properties to enhance the accessibility of dynamic and interactive elements. Use them judiciously when semantic HTML is insufficient.

- **Keyboard Navigation:** Ensuring all interactive elements are navigable using the keyboard (tabindex).
- **Color Contrast:** Understanding WCAG guidelines for sufficient color contrast between text and background.
- **Image Accessibility:** Using the alt attribute to provide textual descriptions for images.

B. CSS (Cascading Style Sheets)

CSS is responsible for the presentation and styling of web pages. Mastery of layout techniques, responsiveness, and performance optimization is key.

- **Layout Techniques:**
  - **Flexbox (Flexible Box Layout):** A powerful one-dimensional layout system for arranging items in rows or columns. Understand flex containers, flex items, properties like flex-direction, justify-content, align-items, flex-grow, flex-shrink, flex-basis. Know when to use it for single-direction layouts and component alignment.
  - **CSS Grid Layout:** A two-dimensional layout system for creating complex grid-based layouts. Understand grid containers, grid items, grid-template-rows, grid-template-columns, grid-area, gap. Know when to use it for overall page structure and complex component layouts.
  - **Choosing Between Flexbox and Grid:** Understand their strengths and weaknesses and when to use each or combine them effectively.
- **Responsive Design:**
  - **Media Queries:** Using @media rules to apply different styles based on screen size, orientation, and other media features.
  - **Mobile-First Approach:** Designing for smaller screens first and then progressively enhancing for larger screens.
  - **Fluid Layouts:** Using relative units (%, em, rem, vw, vh) to create layouts that adapt to different screen sizes.
  - **Viewports:** Understanding the <meta name="viewport"> tag and its importance for controlling the initial scale and width of the viewport.
- **CSS Preprocessors (Sass/Less):**

    - **Features:** Variables, mixins (reusable blocks of CSS declarations), nesting, functions, and more. Understand how they can improve CSS organization and maintainability.
    - **Workflow:** How preprocessor code is compiled into standard CSS.
- **CSS Methodologies (BEM, Atomic CSS):**
    - **BEM (Block, Element, Modifier):** A naming convention for CSS classes that promotes modularity, reusability, and maintainability. Understand the structure and benefits.
    - **Atomic CSS (Functional CSS):** Creating small, single-purpose utility classes. Understand the trade-offs in terms of HTML verbosity and reusability.
- **CSS Performance:**
    - **Critical Rendering Path:** Understanding how the browser processes HTML and CSS to render the page. Optimize the delivery of critical CSS.
    - **Selectors:** Understanding the performance implications of different CSS selectors. Avoid overly specific or complex selectors.
    - **Minification and Compression:** Reducing CSS file sizes for faster loading.
    - **Avoiding !important:** Understand its drawbacks and when it should be absolutely necessary.

C. JavaScript (ES6+ and Beyond)

JavaScript is the language that adds interactivity and dynamic behavior to web pages. A strong grasp of modern JavaScript features and asynchronous programming is crucial.

- **ES6+ Features:**
    - **Arrow Functions:** Concise syntax for function expressions. Understand their lexical this binding.
    - **Destructuring:** Extracting values from arrays and objects into distinct variables.
    - **Spread/Rest Operators:** Expanding iterables and collecting function arguments.
    - **Promises:** Handling asynchronous operations in a more structured way than callbacks. Understand the different states of a promise (pending, fulfilled, rejected) and how to chain them (.then(), .catch(), .finally()).

    - **Async/Await:** Syntactic sugar over promises, making asynchronous code look and behave more like synchronous code. Understand how async functions and the await keyword work.
    - **Modules (import/export):** Organizing JavaScript code into reusable modules. Understand named and default exports/imports.
    - **Classes:** Syntactic sugar over JavaScript's prototype-based inheritance. Understand class definitions, constructors, methods, inheritance (extends), and super.
    - **Template Literals:** String interpolation using backticks.
    - **let and const:** Block-scoped variable declarations, understanding immutability with const.
- **DOM Manipulation:**
    - **Efficient Updates:** Understanding the browser's rendering process and minimizing direct DOM manipulation for better performance. Use techniques like creating document fragments for batch updates.
    - **Event Handling:** Understanding event bubbling and capturing, using event delegation to efficiently handle events on multiple elements.
- **Asynchronous JavaScript:**
    - **Event Loop:** Understanding how JavaScript handles asynchronous tasks using the event loop, call stack, and message queue.
    - **Callbacks:** Traditional way of handling asynchronous operations (and their drawbacks - callback hell).
    - **Promises and Async/Await (as covered in ES6+):** Master these modern approaches.
    - **Error Handling in Asynchronous Operations:** Using .catch() with promises and try...catch with async/await.
- **Testing:**
    - **Unit Testing (Jest, Mocha, Jasmine):** Writing tests for individual functions and components to ensure they behave as expected. Understand concepts like test runners, assertions, mocking, and test coverage.

- **Integration Testing (React Testing Library, Enzyme):** Testing the interaction between different components or modules. Focus on testing user behavior rather than implementation details (React Testing Library).

D. React JS

React is a popular JavaScript library for building user interfaces. A deep understanding of its core concepts, state management, hooks, and performance optimization is essential.

- **Core Concepts:**
  - **Component-Based Architecture:** Building UIs by composing reusable components. Understand functional and class components.
  - **JSX (JavaScript XML):** A syntax extension that allows you to write HTML-like structures within JavaScript.
  - **Virtual DOM:** Understanding how React uses a virtual DOM to efficiently update the actual DOM. Learn about reconciliation and the diffing algorithm.
  - **Props (Properties):** Passing data down from parent to child components. Understand immutability of props.
  - **State:** Managing dynamic data within a component. Understand the difference between useState (for functional components) and this.state and this.setState (for class components). State should be localized as much as possible.
- **Component Lifecycle (for Class Components):**
  - **Mounting:** constructor, static getDerivedStateFromProps, render, componentDidMount.
  - **Updating:** static getDerivedStateFromProps, shouldComponentUpdate, render, getSnapshotBeforeUpdate, componentDidUpdate.
  - **Unmounting:** componentWillUnmount.
  - Understand the purpose and timing of each lifecycle method and when to use them for side effects, data fetching, etc.
- **Hooks (for Functional Components):**
  - **useState:** For adding state to functional components.
  - **useEffect:** For performing side effects (data fetching, subscriptions, timers). Understand the dependency array and cleanup functions.

    - **useContext:** For accessing values from the React Context API.
    - **useReducer:** For more complex state logic, similar to Redux's reducer.
    - **useCallback:** For memoizing callback functions to prevent unnecessary re-renders of child components.
    - **useMemo:** For memoizing expensive computations.
    - **useRef:** For accessing DOM elements or persisting mutable values across renders.
    - **Custom Hooks:** Creating your own reusable logic. Understand the rules of hooks.
- **State Management:**
    - **Local Component State (useState):** Suitable for simple, localized state.
    - **Context API (useContext):** For sharing state that is considered "global" for a subtree of components. Good for themes, user authentication.
    - **Redux:** A predictable state container for complex applications. Understand actions, reducers, store, dispatch, selectors, middleware (e.g., Thunk, Saga). Know its benefits and when its complexity is justified.
    - **Recoil, Zustand, Jotai:** Explore alternative state management libraries and understand their trade-offs (simplicity, performance, scalability).
- **Routing (React Router):**
    - **Core Concepts:** Routes, navigation (<Link>, useNavigate), route parameters.
    - **Different Router Types:** BrowserRouter, HashRouter.
    - **Nested Routes:** Organizing complex UIs with nested layouts.
    - **Route Guards:** Protecting routes based on authentication or other conditions.
- **Performance Optimization:**
    - **Code Splitting (using React.lazy and <Suspense>):** Loading components on demand to reduce initial bundle size.
    - **Memoization (React.memo, useMemo, useCallback):** Preventing unnecessary re-renders of components and functions.
    - **Virtualized Lists (e.g., react-virtualized, react-window):** Efficiently rendering large lists of data.

    - **Profiling Tools (React DevTools):** Identifying performance bottlenecks.
- **Testing (as covered in JavaScript):** Focus on testing React components effectively using React Testing Library.
- **Component Design Principles:**
    - **Reusability:** Creating components that can be used in multiple parts of the application.
    - **Separation of Concerns:** Keeping UI logic, data fetching, and presentation logic separate.
    - **Single Responsibility Principle:** Each component should ideally do one thing well.
- **Server-Side Rendering (SSR) / Next.js:**
    - **Benefits:** Improved SEO, faster initial load time.
    - **Basic Concepts:** How SSR works, the difference between client-side and server-side rendering. Familiarity with Next.js framework is a plus.

II. Backend Development (Core Java, Spring, SpringBoot, Spring Cloud, Java 8 Features, Design Patterns, SSO, JWT, Spring Security)

A. Core Java:

A strong foundation in core Java is essential for any backend role.

- **Object-Oriented Programming (OOP):**
    - **Principles:** Encapsulation (bundling data and methods), Inheritance (creating new classes from existing ones), Polymorphism (objects taking on many forms), Abstraction (hiding complex implementation details). Understand the benefits of each principle.
    - **SOLID Principles:** Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle. Understand how applying these principles leads to maintainable and scalable code.
- **Data Structures and Collections:**
    - **In-depth Understanding:** List (ArrayList, LinkedList), Set (HashSet, TreeSet, LinkedHashSet), Map (HashMap, TreeMap, LinkedHashMap,

ConcurrentHashMap). Understand their internal implementations, time and space complexities for various operations, and when to choose each.

    - **Generics:** Understanding type safety and how generics work in Java.

- **Concurrency:**
    - **Threads:** Creating and managing threads (Runnable, Thread class).
    - **Synchronization:** Mechanisms for controlling access to shared resources (synchronized keyword, synchronized blocks). Understand the risks of race conditions and deadlocks.
    - **Locks:** Explicit locking using java.util.concurrent.locks (e.g., ReentrantLock). Understand fairness and other advanced locking features.
    - **Concurrent Collections:** Thread-safe collections like ConcurrentHashMap, CopyOnWriteArrayList, etc. Understand when to use them over their non-concurrent counterparts.
    - **Thread Pools:** Understanding the benefits of thread pools and how to use ExecutorService. Understand different types of thread pools (fixed, cached, etc.).
- **Exception Handling:**
    - **Best Practices:** Using try-catch-finally blocks effectively. Understanding checked vs. unchecked exceptions and when to throw custom exceptions. Logging exceptions properly.
- **JVM Internals (Basic):**
    - **Memory Management:** Heap (object allocation), Stack (method calls, local variables), Garbage Collection (different algorithms and how they work conceptually).
    - **Class Loading:** How classes are loaded and initialized by the JVM.

B. Spring Framework:

Spring is a comprehensive application development framework for Java.

- **Core Container:**
    - **Dependency Injection (DI):** The concept of providing dependencies to objects instead of them creating them. Understand the benefits (loose coupling, testability).

    - **Inversion of Control (IoC):** The framework controls the lifecycle of objects.
    - **Bean Configuration:** Different ways to configure beans: XML configuration, annotation-based configuration (@Component, @Service, @Repository, @Controller, @Configuration, @Bean), and Java-based configuration. Understand the pros and cons of each.
    - **Bean Scopes:** Singleton, Prototype, Request, Session, Application, WebSocket. Understand the lifecycle of beans in different scopes.
- **Spring MVC (Model-View-Controller):**
    - **Request Handling Lifecycle:** How Spring MVC processes incoming HTTP requests. Understand DispatcherServlet, HandlerMapping, HandlerAdapter, ViewResolver.
    - **Controllers:** Handling incoming requests using @Controller and @RestController.
    - **Request/Response Mapping:** Using annotations like @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @RequestMapping, @PathVariable, @RequestParam, @RequestBody, @ResponseBody.
    - **Data Binding:** How Spring MVC automatically converts request parameters and body to Java objects.
    - **Validation:** Using @Valid and javax.validation.constraints for validating request data.
- **Spring Data JPA (Java Persistence API):**
    - **Working with Databases:** Simplifying database access using JPA.
    - **Repositories:** Using JpaRepository and defining custom query methods. Understand different ways to write queries (@Query, derived query methods).
    - **ORM (Object-Relational Mapping) Concepts:** Understanding how Java objects are mapped to database tables.
    - **Transactions:** Managing database transactions using @Transactional. Understand different propagation levels.
- **Aspect-Oriented Programming (AOP):**
    - **Concepts:** Aspects, Advice (e.g., @Before, @After, @Around), Pointcuts, Join Points.

    - **Use Cases:** Logging, transaction management, security, cross-cutting concerns. Understand how AOP can decouple cross-cutting concerns from core business logic.

- **Spring Boot:**
    - **Auto-Configuration:** How Spring Boot automatically configures your application based on dependencies.
    - **Starters:** Dependency descriptors that simplify adding common dependencies.
    - **Actuator:** Providing production-ready features like health checks, metrics, and more.
    - **Spring Boot CLI:** Command-line tool for rapid application development.
- **Spring Cloud (Fundamentals):**
    - **Service Discovery (e.g., Eureka, Consul):** How microservices register and discover each other. Understand client-side and server-side discovery patterns.
    - **API Gateway (e.g., Zuul, Spring Cloud Gateway):** A single entry point for external requests to microservices. Understand routing, filtering, and other gateway functionalities.
    - **Configuration Management (e.g., Spring Cloud Config):** Centralizing and versioning application configuration.
    - **Circuit Breaker (e.g., Hystrix, Resilience4j):** Preventing cascading failures in distributed systems. Understand the concepts of open, closed, and half-open states.
    - **Load Balancing (e.g., Ribbon, LoadBalancer):** Distributing incoming requests across multiple instances of a service. Understand different load balancing algorithms.

C. Java 8 Features (Continued):

- **Lambda Expressions:** Concise syntax for anonymous functions. Understand functional interfaces and how lambda expressions can implement them.
- **Streams API:** A powerful way to process collections of data in a declarative and efficient manner. Understand stream operations like filter, map, reduce, collect, forEach, sorted, distinct, etc. Learn about intermediate and terminal operations.

- **Optional:** A container object that may or may not contain a non-null value. Understand how to use Optional to avoid NullPointerException and write more robust code.
- **Date and Time API (java.time package):** Understand the new date and time classes (LocalDate, LocalTime, LocalDateTime, ZonedDateTime, Period, Duration) and how they address the shortcomings of the legacy java.util.Date and Calendar classes.
- **Method References:** A concise way to refer to existing methods by their name. Understand different types of method references (static method, instance method of a particular object, instance method of an arbitrary object of a particular type, constructor reference).

D. Design Patterns (GoF - Gang of Four):

A solid understanding of fundamental design patterns is crucial for writing maintainable, reusable, and flexible code.

- **Creational Patterns:** Focus on how objects are created.
    - **Singleton:** Ensures that a class has only one instance and provides a global point of access to it. Understand different implementation approaches (eager, lazy, thread-safe).
    - **Factory Method:** Defines an interface for creating an object, but lets subclasses decide which class to instantiate.
    - **Abstract Factory:** Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
    - **Builder:** Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
    - **Prototype:** Specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype.
- **Structural Patterns:** Deal with the composition of classes and objects.
    - **Adapter:** Converts the interface of a class into another interface clients expect.
    - **Bridge:** Decouples an abstraction from its implementation so that the two can vary independently.
    - **Composite:** Composes objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions uniformly.
    - **Decorator:** Dynamically adds responsibilities to an object.

    - **Facade:** Provides a unified interface to a set of interfaces in a subsystem.
    - **Flyweight:** Uses sharing to support large numbers of fine-grained objects efficiently.
    - **Proxy:** Provides a surrogate or placeholder for another object to control access to it.
- **Behavioral Patterns:** Concern algorithms and the assignment of responsibilities between objects.
    - **Chain of Responsibility:** Avoids coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request.
    - **Command:** Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
    - **Interpreter:** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
    - **Iterator:** Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
    - **Mediator:** Defines an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly.
    - **Memento:** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
    - **Observer:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
    - **State:** Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
    - **Strategy:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently of the clients that use it.

    - **Template Method:** Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
    - **Visitor:** Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- **Focus:** Understand the intent, motivation, applicability, and consequences of each pattern. Be able to recognize when a pattern might be applicable in a given scenario and discuss its trade-offs.

E. Single Sign-On (SSO):

Understanding SSO is important for secure and user-friendly authentication in modern applications.

- **Concepts:** Centralized authentication where users log in once and gain access to multiple applications without re-authenticating. Understand the benefits like improved user experience, reduced administrative overhead, and enhanced security.
- **Protocols:**
    - **SAML (Security Assertion Markup Language):** An XML-based open standard for exchanging authentication and authorization data between an identity provider (IdP) and a service provider (SP). Understand the basic flow involving assertions and requests/responses.
    - **OAuth 2.0 (Open Authorization):** An authorization framework that enables third-party applications to obtain limited access to an HTTP service. Understand the different grant types (authorization code, implicit, client credentials, resource owner password credentials) and when to use each.
    - **OpenID Connect (OIDC):** An authentication layer on top of OAuth 2.0. It provides a standardized way to verify the identity of end-users. Understand the concept of ID tokens and how they carry user identity information.

F. JSON Web Tokens (JWT):

JWTs are a common way to securely transmit information between parties as a JSON object.

- **Structure:** Understand the three parts of a JWT: Header (metadata about the token), Payload (claims or information about the user and other data), and Signature (used to verify the integrity of the token).

- **Usage:** Primarily used for authentication (sending user identity after login) and authorization (verifying if a user has permission to access a resource). Understand how JWTs are typically used in stateless backend architectures.
- **Security Considerations:** Understand the importance of using strong signing algorithms (e.g., HS256, RS256), keeping the secret key secure, setting appropriate expiration times, and being aware of potential vulnerabilities.

G. Spring Security:

Spring Security is a powerful and highly customizable framework for providing authentication and authorization to Java applications.

- **Authentication:** How Spring Security identifies users. Understand concepts like Principal, AuthenticationManager, AuthenticationProvider, and different authentication mechanisms (e.g., basic authentication, form-based login, OAuth 2.0/OIDC integration).
- **Authorization:** How Spring Security controls access to resources. Understand concepts like GrantedAuthority (roles, permissions), AccessDecisionManager, AccessDecisionVoter, and different ways to define authorization rules (e.g., method security annotations like @PreAuthorize, @PostAuthorize).
- **Security Filters:** Understand the FilterChain in Spring Security and the purpose of various built-in filters (e.g., BasicAuthFilter, UsernamePasswordAuthenticationFilter, JwtAuthenticationFilter).
- **OAuth 2.0 and OpenID Connect Integration:** How Spring Security can be configured as an OAuth 2.0 client or resource server, and how it handles OIDC flows and ID token verification.

III. Microservices (Microservices Architecture, Design Patterns)

A. Microservices Architecture:

Understanding the principles and trade-offs of microservices is crucial for modern backend development.

- **Principles:**
    - **Single Responsibility:** Each service focuses on a specific business capability.
    - **Autonomy:** Services can be developed, deployed, and scaled independently.
    - **Decentralized Governance:** Different teams can choose the best technologies for their services.

    - **Design for Failure:** Services should be resilient and handle failures gracefully.
- **Benefits and Drawbacks:** Understand the advantages (scalability, maintainability, technology diversity, faster development cycles) and disadvantages (complexity, operational overhead, distributed debugging, network latency).
- **Communication Patterns:**
    - **REST (Representational State Transfer):** A widely used architectural style for building web services. Understand statelessness, resource-based URLs, and HTTP methods.
    - **gRPC:** A modern, open-source, high-performance Remote Procedure Call (RPC) framework. Understand its use of Protocol Buffers for serialization and HTTP/2 for transport.
    - **Message Queues (e.g., Kafka, RabbitMQ):** Asynchronous communication between services. Understand concepts like producers, consumers, topics, queues, and different messaging patterns.
    - **Choosing Between Communication Patterns:** Understand the trade-offs between synchronous (REST, gRPC) and asynchronous (message queues) communication in terms of coupling, reliability, and performance.
- **Data Management:**
    - **Database per Service:** Each microservice has its own database, promoting data isolation and autonomy.
    - **Shared Database (Challenges):** Understand the drawbacks of sharing a database across multiple microservices (tight coupling, risk of conflicts).
    - **Data Consistency in Distributed Systems:** Explore patterns like Saga for managing distributed transactions.
- **Service Discovery:** How services locate each other in a dynamic environment. Understand client-side (e.g., Netflix Eureka) and server-side (e.g., Consul) discovery mechanisms.
- **API Gateway:** A single entry point for external clients to access microservices. Understand its responsibilities like routing, authentication, rate limiting, and request transformation.
- **Containerization (Docker):** Essential for packaging and deploying microservices consistently across different environments.

- **Orchestration (Kubernetes):** A platform for automating deployment, scaling, and management of containerized applications.

B. Microservices Design Patterns:

Specific design patterns address the challenges of building distributed systems.

- **API Gateway:** Provides a single entry point for clients, simplifying the client experience and allowing for cross-cutting concerns to be handled centrally.
- **Circuit Breaker:** Prevents cascading failures by stopping requests to a failing service for a period of time.
- **Service Discovery:** Enables services to find and communicate with each other without hardcoding addresses.
- **Externalized Configuration:** Managing application configuration externally (e.g., using Spring Cloud Config or HashiCorp Vault) for easier updates and environment-specific settings.
- **Log Aggregation:** Centralizing logs from all microservices for easier monitoring and debugging.
- **Distributed Tracing:** Tracking requests as they propagate through multiple services to identify performance bottlenecks and issues.
- **Event-Driven Architecture:** Using events to trigger actions in other services, promoting loose coupling and asynchronicity.
- **Saga Pattern:** Managing long-lived, distributed transactions across multiple services by coordinating local transactions. Understand different saga implementation strategies (choreography, orchestration).

IV. DevOps (Docker, Kubernetes, AWS, ArgoCD, Openshift)

A. Docker:

Containerization is a fundamental aspect of modern software deployment.

- **Containers vs. Virtual Machines:** Understand the differences in terms of resource utilization, startup time, and isolation.
- **Docker Images:** How images are built using Dockerfiles (understand instructions like FROM, RUN, COPY, ADD, ENV, EXPOSE, CMD, ENTRYPOINT). Learn about image layers and optimization techniques.

- **Docker Compose:** Defining and managing multi-container Docker applications using YAML files. Understand services, networks, and volumes.
- **Docker Networking:** Different networking modes in Docker (bridge, host, none, custom networks) and how containers communicate with each other and the outside world.
- **Docker Volumes:** Persisting data generated by containers beyond the container's lifecycle. Understand different types of volumes (named volumes, bind mounts).

B. Kubernetes (K8s):

Kubernetes is a powerful container orchestration platform.

- **Core Concepts:**
    - **Pods:** The smallest deployable unit in Kubernetes, containing one or more containers.
    - **Services:** An abstraction that exposes a set of Pods as a single network endpoint. Understand different service types (ClusterIP, NodePort, LoadBalancer).
    - **Deployments:** A declarative way to manage ReplicaSets, ensuring a specified number of Pod replicas are running. Understand rolling updates and rollbacks.
    - **StatefulSets:** Manages the deployment and scaling of stateful applications with stable network identities and persistent storage.
    - **Namespaces:** Logical isolation within a Kubernetes cluster.
    - **Volumes:** Providing persistent storage to Pods. Understand different volume types (e.g., persistent volumes, config maps, secrets).
    - **ConfigMaps:** Storing non-confidential configuration data for applications.
    - **Secrets:** Storing sensitive information like passwords and API keys.
- **Architecture:** Understand the roles of the control plane components (API server, etcd, scheduler, controller manager) and worker node components (kubelet, kube-proxy, container runtime).
- **kubectl:** Essential command-line tool for interacting with a Kubernetes cluster. Know common commands for managing pods, deployments, services, etc.
- **Scaling:**
    - **Horizontal Pod Autoscaler (HPA):** Automatically scales the number of Pod replicas based on metrics like CPU utilization or custom metrics.

    - **Vertical Pod Autoscaler (VPA):** Automatically adjusts the CPU and memory resources allocated to Pods.
- **Networking:** Service discovery within the cluster, ingress controllers for exposing services externally.
- **Helm:** A package manager for Kubernetes, simplifying the deployment and management of applications.

C. AWS (Amazon Web Services):

Familiarity with cloud platforms is increasingly important.

- **Compute (EC2 - Elastic Compute Cloud):** Understand different instance types, security groups for network security, auto-scaling groups for automatic scaling of instances.
- **Storage (S3 - Simple Storage Service, EBS - Elastic Block Store, RDS - Relational Database Service):** Understand object storage, block storage, and managed relational databases. Know when to use each.
- **Containers (ECS - Elastic Container Service, EKS - Elastic Kubernetes Service):** AWS's managed container orchestration services. Understand the differences between them.
- **Networking (VPC - Virtual Private Cloud, Subnets, Security Groups, ELB - Elastic Load Balancing):** Understand how to create and manage virtual networks, control network access, and distribute traffic.
- **Serverless (Lambda, API Gateway):** Basic understanding of serverless computing and how it can be used to build scalable applications.

D. ArgoCD:

A popular GitOps tool for continuous delivery in Kubernetes.

- **GitOps:** Understand the principles of managing infrastructure and application configurations declaratively in Git.
- **Declarative Configuration:** Applications and infrastructure are defined in Git repositories.
- **Continuous Delivery:** ArgoCD automatically deploys changes to Kubernetes clusters based on the state in Git.
- **Application CRD (Custom Resource Definition):** Understand how ArgoCD manages applications as Kubernetes custom resources.

E. Openshift:

Red Hat's enterprise Kubernetes platform.

- **Kubernetes Distribution:** Understand that Openshift is built on top of Kubernetes but provides additional features and a more opinionated approach.
- **Key Differences from Vanilla Kubernetes:** Focus on developer experience, integrated CI/CD pipelines (Tekton), enhanced security features (Security Context Constraints - SCCs), and a more integrated platform.

V. System Design (Fundamentals, Design like FB, Uber, TinyUrl etc, HLD, LLD)

A. Fundamentals:

Understanding core principles of scalable and reliable systems.

- **Scalability:**
    - **Horizontal Scaling:** Adding more machines to handle increased load.
    - **Vertical Scaling:** Upgrading the resources (CPU, RAM, storage) of a single machine. Understand the trade-offs of each approach.
- **Reliability:** Designing systems to be fault-tolerant and continue functioning even when components fail. Understand concepts like redundancy, replication, and failover.
- **Availability:** The percentage of time a system is operational and accessible. Understand SLOs (Service Level Objectives) and SLAs (Service Level Agreements).
- **Consistency:** How data is kept up-to-date across different parts of a distributed system. Understand different consistency models (strong consistency, eventual consistency) and their trade-offs.
- **Performance:** Optimizing for low latency (response time) and high throughput (number of requests processed per unit of time). Understand the impact of network latency and bottlenecks.
- **Security:** Designing systems with security in mind to protect data and prevent unauthorized access. Understand common security threats (e.g., SQL injection, XSS) and mitigation strategies.
- **CAP Theorem:** Understanding the fundamental trade-offs between Consistency, Availability, and Partition Tolerance in distributed systems.

B. Design Case Studies:

Practice designing popular systems to apply the fundamentals.

- **Facebook News Feed:** Consider data ingestion, storage, ranking algorithms, real-time updates, and scalability.
- **Uber Ride-Sharing:** Think about real-time location tracking, matching algorithms, payment processing, and handling concurrent requests.
- **TinyURL:** Focus on URL shortening, redirection mechanisms, and handling high traffic.
- **Twitter:** Consider tweet ingestion, storage, fan-out mechanisms, and real-time updates.
- **Instagram:** Think about image/video storage, feed generation, and handling large numbers of users.
- **Approach:** For each case study, identify the core requirements, constraints (scale, latency, consistency), and then propose a high-level architecture with key components and their interactions. Discuss trade-offs and potential bottlenecks.

C. High-Level Design (HLD):

Focuses on the overall architecture of a system.

- **Identifying Key Components:** Breaking down the system into major functional modules or services.
- **Data Flow:** Understanding how data moves between different components.
- **API Design:** Defining the interfaces (REST, gRPC) between services or between the client

Sources