KOENIG

step forward

Full Stack Advanced Training in Java

Module 1: Advanced Java Core Concepts

Module 1: Advanced Java Core Concepts

- Deep dive into Java 8+ features:
  - Java 8 features (example: Lambda, Stream API, Functional Interface)
  - Java 21 features (example: Record, Virtual Thread, Pattern Matching etc.)
- Java Memory Model (Garbage Collection, Heap, Stack)
- Best Practices for Exception Handling
- JVM Optimization Techniques

Java 8 features

- Lambda Expressions
  - A lambda expression is a short block of code which takes in parameters and returns a value.
  - Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.
- Syntax:

```
(parameters) -> expression
or
(parameters) -> { statements }
```

Java 8 features

- Functional Interfaces
  - An interface with exactly one abstract method, used as the target for lambda expressions.
  - You can annotate them to clarify purpose with @FunctionalInterface
- Stream API
  - The Streams API is a powerful way to process collections of data using functional programming principles.
  - It allows you to perform filtering, mapping, reducing, and more.
  - Key Features of Streams
    - Lazy Evaluation: Operations are not performed until a terminal operation is invoked.
    - Immutability: Streams do not modify the original data source.
    - Pipelining: Multiple operations can be chained together.

Java 8 features

- Method References
  - A shorter form of lambda when calling an existing method.
  - Example:

    ```
    List<String> names = Arrays.asList("Tom", "Jerry", "Spike");
    names.forEach(System.out::println);
    ```
  - Equivalent to:

    ```
    names.forEach(name -> System.out.println(name));
    ```
- Default Methods in Interfaces
  - Add methods with implementations inside interfaces (without breaking old code).

Java 21 features

- Records
  - Records are a special kind of Java class designed to hold immutable data with minimal boilerplate code.
  - They are ideal for DTOs (Data Transfer Objects) or simple value carriers in Spring Boot and other Java applications.
  - Purpose of Records
    - To simplify POJOs with getters, constructor, equals(), hashCode(), and toString() — all auto-generated.
    - Promote immutability
    - Ideal for API responses, request models, or configuration values

Java 21 features

- Pattern Matching
  - Pattern matching simplifies conditional logic when checking object types and extracting values in if, switch, and other constructs — without verbose casting.
  - Makes code:
    - More concise
    - More readable
    - Type-safe
  - Currently supported in:
    - instanceof checks (Java 16+)
    - switch expressions with patterns (Java 21+)

Java 21 features

- Virtual Threads
  - Virtual threads are lightweight threads that allow your Java applications to handle thousands or millions of concurrent tasks efficiently — like HTTP requests, database calls, or message processing.
  - Virtual threads are ideal for:
    - High-concurrency web servers
    - Asynchronous Spring Boot apps
    - Replacing thread pools with simpler code

Java Memory Model (Garbage Collection, Heap, Stack)

- The Java Memory Model (JMM) defines how variables are read and written when multiple threads access shared memory.
- Java Memory Model ensures:
  - Visibility of shared data between threads
  - Ordering of read/write operations
  - Safe concurrent programming

Java Memory Model (Garbage Collection, Heap, Stack)

- Garbage Collection
  - Garbage Collection is the process of automatically reclaiming memory by removing objects that are no longer reachable.
  - You don’t need to manually delete objects in Java — the JVM’s garbage collector (GC) handles it.
- How Garbage Collection Works
  1. Java allocates memory in the heap.
  2. The JVM identifies unreachable objects (no reference pointing to them).
  3. These objects are automatically deleted to free up memory.

Java Memory Model (Garbage Collection, Heap, Stack)

- Garbage Collection
  - Garbage Collection is the process of automatically reclaiming memory by removing objects that are no longer reachable.
  - You don’t need to manually delete objects in Java — the JVM’s garbage collector (GC) handles it.
- How Garbage Collection Works
  1. Java allocates memory in the heap.
  2. The JVM identifies unreachable objects (no reference pointing to them).
  3. These objects are automatically deleted to free up memory.

Java Memory Model (Garbage Collection, Heap, Stack)

- When you run a Java program, the JVM divides memory into several regions.
- The two most important is Heap Memory & Stack Memory.
- Simple Real-World Analogy
  - Stack → Stack of Plates:
    - You push (call method), then pop (method exits).
    - Only the top method is active.
  - Heap → Big Warehouse:
    - You store objects (like Spring beans).
    - Anyone (thread) can access and use it until the object is removed.

Java Memory Model (Garbage Collection, Heap, Stack)

- Heap Memory
  - Allocated for dynamic memory (objects, beans, arrays).
  - Shared memory area used by all threads.
  - Stores Objects (new MyClass()), Instance variables (fields of objects) & Arrays
- Example:
  - String s = new String("Hello");
  - The String object is stored in the Heap.
  - The reference s is stored in the Stack

Java Memory Model (Garbage Collection, Heap, Stack)

- Stack Memory
  - Each thread has its own stack, separate from other threads.
  - Stores Method call frames, Local variables, References to objects in the Heap & Return addresses
  - Key Characteristics:
    - Follows LIFO (Last In, First Out) order.
    - Memory is automatically freed when the method exits (no GC needed).
    - Much faster access than the Heap.
  - Example:

```
public void doWork() {
    int a = 5;        // stored in Stack
    MyClass obj = new MyClass(); // 'obj' reference stored in Stack, object in Heap
}
```

Best Practices for Exception Handling

- Catch Only What You Can Handle
  - Don’t catch exceptions just to suppress them. Only catch if you can meaningfully recover or log.
  - Bad Example:

```
try {
    someOperation();
} catch (Exception e) {
    // silently ignore
}
```

  - Good Example:

```
try {
    someOperation();
} catch (IOException e) {
    logger.error("I/O failure", e);
    throw e; // or recover accordingly
}
```

Best Practices for Exception Handling

- Use Specific Exception Types
  - Catching broad exceptions (Exception, Throwable) hides real problems.
- Example

```
try {
    readFile();
} catch (FileNotFoundException e) {
    // handle missing file
} catch (IOException e) {
    // handle other I/O issues
}
```

Best Practices for Exception Handling

- Never Swallow Exceptions Without Logging or Action
  - Always log or rethrow. Silent failure is a debugging terrible.
- Example:
  - Use logging frameworks like SLF4J + Logback:

```
logger.warn("Validation failed", e);
```

Best Practices for Exception Handling

- Clean Up Resources in finally or Try-with-Resources
  - Always release resources to prevent leaks.
  - Try-with-resources (Java 7+):

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String line = br.readLine();
}
```

Best Practices for Exception Handling

- Don’t Use Exceptions for Control Flow
  - They are expensive and confusing if used as normal logic.
  - Bad Example:

```
try {
    while (true) {
        nextElement();
    }
} catch (NoSuchElementException e) {
    // end of iteration
}
```

  - Better:

```
while (iterator.hasNext()) {
    iterator.next();
}
```

Best Practices for Exception Handling

- Create Custom Exceptions When Applicable
  - Domain-specific exceptions clarify meaning and improve maintainability.
  - Example:

```
public class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

  - Throwing:

```
if (balance < withdrawal) {
    throw new InsufficientFundsException("Balance too low");
}
```

Best Practices for Exception Handling

- Preserve the Cause
  - Always chain the original exception to avoid losing stack trace.
  - Incorrect:

```
catch(SQLException e) {
    throw new DataAccessException("Error accessing data");
}
```

  - Correct:

```
catch(SQLException e) {
    throw new DataAccessException("Error accessing data", e);
}
```

Best Practices for Exception Handling

- Use Checked vs Unchecked Exceptions Properly
  - Checked Exception (extends Exception):
    - Represent recoverable conditions (e.g., IOException, ParseException).
    - Caller must handle or declare.
  - Unchecked Exception (extends RuntimeException):
    - Programmer errors (e.g., NullPointerException, IllegalArgumentException).
    - Usually not recoverable.

Best Practices for Exception Handling

- Fail Fast, Validate Early
  - Check input parameters at the beginning and throw appropriate exceptions.
  - Example:

```
public void transferMoney(Account from, Account to, double amount) {
    if (from == null || to == null) {
        throw new IllegalArgumentException("Accounts cannot be null");
    }
    if (amount <= 0) {
        throw new IllegalArgumentException("Amount must be positive");
    }
    // proceed
}
```

Best Practices for Exception Handling

- Use Optional Instead of Returning Null
  - Avoid null where possible:

```
public Optional<Customer> findCustomer(String id) {
   // return Optional.of(customer) or Optional.empty()
}
```

  - Caller:

```
customer.findCustomer(id).ifPresent(c -> process(c));
```

JVM Optimization Techniques

- Memory Management Optimization
  - Balance memory allocation, GC frequency, and latency.
- Heap Size Tuning
  - -Xms: Initial heap size
  - -Xmx: Maximum heap size
- Example:

```
-Xms1g -Xmx4g
```

JVM Optimization Techniques

- Garbage Collection Tuning
  - Choose the GC algorithm that fits your workload.
  - Common GCs and When to Use Them

GC Algorithm	Use Case
Serial	Small apps, single-threaded environments
Parallel	Throughput-focused batch processing
CMS (deprecated)	Low-pause applications
G1 (default)	Balanced throughput and low pauses
ZGC/Shenandoah	Ultra-low latency, massive heaps

  - Example: Enabling G1 GC

```
-XX:+UseG1GC
```

JVM Optimization Techniques

- JIT Compilation & Profiling
  - Java uses Just-In-Time (JIT) compilation to optimize performance.
- Tiered Compilation
  - Modern JVMs compile code at multiple optimization levels:

```
-XX:+TieredCompilation
```

- Ahead-of-Time Compilation (AOT)
  - Since Java 9, you can pre-compile code:

```
jaotc --output libHelloWorld.so HelloWorld.class
```

  - Note: Rarely used in most applications, but important for faster startup.

JVM Optimization Techniques

- Classloading Optimization
  - Avoid unnecessary class loading and reflection—especially in microservices.
- Class Data Sharing (CDS)
  - Preload commonly used classes to speed up startup:

```
-XX:+UseAppCDS
```

JVM Optimization Techniques

- Monitoring & Observability
  - Measure before you tune.
  - JVM Metrics:
    - Heap usage
    - GC times
    - Thread counts
    - CPU utilization
  - Java Flight Recorder:
    - Capture detailed runtime information:

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

Module 2: Backend Development with Spring Framework

Module 2: Backend Development with Spring Framework

- Spring Core: Spring AOP and Transaction Management
- Spring Boot Persistence Layer:
  - Spring Data JPA (CRUD Operations, JPQL, Native Queries)
- Spring Security:
  - Spring Security (Authentication, Authorization)
  - JWT Implementation for Token-based Security
- Integration:
  - Connecting to external APIs using Asynchronously (ex: Web Client)
  - Messaging with Kafka
- Monitoring:
  - Integrating Actuator and Micrometer for application monitoring

Spring Core

Spring AOP and Transaction Management

- Aspect-Oriented Programming (AOP) lets you separate cross-cutting concerns like: Logging, Security, Transactions & Caching
- Aspect: A class containing cross-cutting logic.
- Advice: The action taken (e.g., “before method runs”).
- Pointcut: Where to apply the advice (e.g., “all methods in package xyz”).

Spring AOP and Transaction Management

- In Spring, a transaction is a set of operations that must succeed or fail as a unit.
  - All succeed = commit
  - Any fail = rollback
- Spring manages transactions declaratively with **@Transactional**.

Spring Boot Persistence Layer

Spring Data JPA (CRUD Operations, JPQL, Native Queries)

- Spring Data JPA simplifies data access layers.
- Automatically implements repositories
- No boilerplate CRUD code
- Supports JPQL and native SQL queries
- Works seamlessly with Hibernate/JPA
- JPQL uses entity names and fields, not table columns

Employee Record Management System

- Build a system to manage employee details in a company using Spring Boot + Spring Data JPA.
- Entity:
  - Employee (id, name, email, department, salary)
- Functional Requirements:
  - CRUD Operations
  - JPQL Queries
    - Get all employees in a specific department
    - Get all employees with salary greater than a given amount
  - Native Queries
    - Count total number of employees

Employee Record Management System

- Create REST API to manage employee details in a company using Spring Boot + Spring Data JPA with MySQL.
- Entity:
  - Employee (id, name, email, department, salary)
- Functional Requirements:
  - CRUD Operations
  - JPQL Queries
    - Get all employees in a specific department
    - Get all employees with salary greater than a given amount
  - Native Queries
    - Count total number of employees

Spring Security

Spring Security (Authentication, Authorization)

- Spring Security is a powerful, customizable framework that handles:
  - Authentication (login, identity verification)
  - Authorization (access control)
  - Password encoding
- Authentication
  - Authentication is about verifying identity.
  - Examples:
    - Username & password (most common)
    - JWT tokens
- Authorization
  - Authorization is about permissions.
  - Example permission roles are USER, ADMIN

JWT Implementation for Token-based Security

- What is JWT (JSON Web Token)?
  - JWT is a secure, compact token used for stateless authentication in REST APIs.
- Structure of a JWT Token: Header.Payload.Signature
  - Header: Contains token type (JWT) and signing algorithm (e.g., HS256).
  - Payload: Contains claims (user details, roles, expiration time, etc.).
  - Signature: Ensures the token is valid and untampered.

Integration

Connecting to external APIs using Asynchronously (ex: Web Client)

- WebClient is part of the Spring WebFlux module.
- It supports reactive, non-blocking HTTP calls.
- You can use it even in non-reactive Spring Boot apps—it doesn't force your whole app to be reactive.
- Suitable for high-concurrency scenarios where you don't want threads blocked waiting for HTTP responses.

Messaging with Kafka

- Apache Kafka is a distributed event streaming platform used for:
  - Publishing and subscribing to streams of records (messages)
  - Real-time processing
  - Building event-driven architectures
- Kafka is widely used for microservices communication, log aggregation, and stream processing.

Messaging with Kafka

- How Messaging Works in Kafka
  - Producer: Sends messages (events) to a Kafka topic.
  - Topic: A named stream where messages are stored (temporarily).
  - Broker: Kafka server that stores and serves messages.
  - Consumer: Reads messages from a topic.
- Kafka in a Spring Boot App
  - Producer: Sends events (e.g., new order created)
  - Consumer: Listens to events (e.g., notification service reacts)

Monitoring

Integrating Actuator and Micrometer for application monitoring

- Spring Boot Actuator
  - Provides production-ready endpoints to monitor and manage your app:
    - /actuator/health
    - /actuator/metrics
    - /actuator/info
    - Many more
- Micrometer
  - A metrics facade that collects and publishes metrics to monitoring systems (Eg. Prometheus).
  - It is embedded in Actuator, so by default you get Micrometer metrics exposed.

Module 3: Frontend Development with Modern JavaScript Frameworks

Module 3: Frontend Development with Modern JavaScript Frameworks

- Introduction to Frontend Frameworks:
  - HTML5 Advance topics (Local storage, Responsive images, Web workers, Drag and Drop API, Web sockets, Microdata etc.)
  - Knowledge of Advance JS and ES6+ features in depth
  - Knowledge of Typescript in depth.
  - Knowledge of React in depth.
  - Basic knowledge of Virtual DOM.
  - Design patterns (Singleton, Factory pattern, Module, Prototype etc.)
  - Micro frontend architecture

Module 3: Frontend Development with Modern JavaScript Frameworks

- Component-based Architecture:
  - Creating Reusable Components
  - Creating Custom Directives.
  - State Management (Redux)
- Routing and Navigation
  - Handling REST API calls (Axios)
  - Real-time Updates with WebSockets

Module 3: Frontend Development with Modern JavaScript Frameworks

- Styling:
  - Deep drive knowledge in CSS3 (Grid, Flex etc.)
  - CSS Preprocessors (SASS/LESS)
  - Bootstrap
- Build Tools: Vite or Webpack
- Unit Testing: Jest

Introduction to Frontend Frameworks

Introduction to Frontend Frameworks

- A frontend framework is a structured collection of prewritten JavaScript (and sometimes HTML/CSS) Tools and libraries.
- Make it easier to build interactive user interfaces for web applications.
- Common Features in Frontend Frameworks
  - Components: Reusable UI pieces (e.g., Card, Navbar)
  - Data Binding: Sync UI with data
  - Routing: Navigate between views
  - State Management: Keep data consistent across components
  - CLI Tools: Generators and build tools

HTML5 Advance topics - Local storage

- Local Storage is part of the Web Storage API.
- It lets you store key-value pairs in the browser persistently (survives page reloads and browser restarts).
- Key Features:
  - Up to ~5–10MB of storage per origin
  - Data stored as strings
- Basic API Methods:
  - localStorage.setItem('key', 'value');   // Save
  - let value = localStorage.getItem('key'); // Read
  - localStorage.removeItem('key');   // Delete
  - localStorage.clear();   // Clear all keys

HTML5 Advance topics - Responsive images

- Responsive images adapt image loading to the screen size, resolution, and device capabilities. This helps:
  - Optimize performance (smaller files for mobile)
  - Improve UX (faster loads, crisp images)
- The srcset and sizes Attributes tells the browser, which images to choose at different resolutions or viewport widths.

HTML5 Advance topics - Web workers

- Web Workers let you run JavaScript in background threads, separate from the main UI thread.
- Why this is useful:
  - JavaScript is single-threaded by default (everything runs on the main thread).
  - Long-running tasks (e.g., complex calculations, data parsing, heavy loops) block the UI, making the app feel frozen.
  - Web Workers let you offload those tasks so the UI stays responsive.
- They do not have access to the DOM, but can communicate with the main thread via messages.
- You must serve files over HTTP (not file://) in most browsers due to security policies

HTML5 Advance topics - Drag and Drop API

- HTML5 Drag and Drop API is a native browser API that allows users to click, drag, and drop elements.
- It works by:
  - Making elements draggable.
  - Listening to drag events.
  - Handling drop operations.

HTML5 Advance topics - Web sockets

- WebSocket is a protocol that provides full-duplex, bidirectional communication between the client (browser) and server over a single, long-lived connection.
- Unlike HTTP:
  - HTTP = request–response (client initiates every exchange)
  - WebSocket = persistent connection (either side can send messages any time)
- Benefits:
  - Low latency (no repeated HTTP handshakes)
  - Lightweight frames (no big headers)
  - Ideal for live updates

HTML5 Advance topics - Microdata

- Microdata is a way to embed machine-readable metadata into HTML using attributes.
- It allows you to label elements with specific types and properties from a vocabulary, most commonly Schema.org.
- This makes your content easier to parse and index by search engines.
- Microdata attributes

Attribute	What it does
itemscope	Declares that this element is an item
itemtype	Specifies the type (e.g., Product, Person)
itemprop	Defines a property of the item

Knowledge of Advance JS and ES6+ features in depth

- LET, CONST, and Block Scoping
  - **let and const** are block-scoped.
  - **const** means the variable binding cannot be reassigned
- Arrow Functions
  - Shorter syntax for anonymous functions
- Template Literals
  - String uses backticks, embed variable value and multiline.
- De-structuring
  - Extract values from arrays and objects

Knowledge of Advance JS and ES6+ features in depth

- Spread and Rest
  - Spread (expand) and Rest (gather)
- Default Parameters
- Classes
- Modules (import/export)

Knowledge of Typescript in depth

- TypeScript is a superset of JavaScript(JS) that adds static typing to JS.
- It was developed by Microsoft to help developers write safer, more maintainable code for large-scale applications.
- Key Features of TypeScript
  - Static Typing
    - You declare types for variables, function arguments, and return values.
  - Type Inference
    - TypeScript can automatically infer types even when not explicitly defined.
  - Interfaces
    - Define contracts for objects and classes.
  - Compile-Time Checks
    - Catch errors before runtime by compiling (transpiling from .ts to .js).
  - Tooling Support
    - Works great with VS Code, providing IntelliSense, autocomplete, and refactoring tools.

Knowledge of React in depth

- React.js is a JavaScript library developed by Facebook (now Meta) for building user interfaces, especially single-page applications (SPAs).
- It lets you build complex UIs by composing small, reusable pieces of code called components.
- Key Features of React
  - Virtual DOM
  - JSX for templating
  - Uses JavaScript or TypeScript
  - Declarative and component-based
  - Unidirectional data flow
  - Rich ecosystem (React Router, Redux, Next.js)

Basic knowledge of Virtual DOM

- The Virtual DOM (VDOM) is a lightweight in-memory representation of the real DOM (Document Object Model).
- React uses it to optimize UI updates and improve performance.
- Virtual DOM technique is used in React library.
- React keeps a lightweight copy of the real DOM in memory.
- When state changes, it creates a new Virtual DOM tree.
- Compares old and new trees (diffing).
- Calculates the minimal set of changes.
- Updates the real DOM efficiently.

Design patterns (Singleton, Factory pattern, Module, Prototype etc.)

- Singleton Pattern
  - Ensure only one instance of a class exists, and provide a global access point to it.
  - Common Use Case:
    - Config objects
    - Database connections
- Factory Pattern
  - Create objects without exposing the instantiation logic to the client.
  - You call a factory method, and it returns an object of a desired type.
  - Common Use Case:
    - When you need to create different types of objects that share an interface.

Design patterns (Singleton, Factory pattern, Module, Prototype etc.)

- Module Pattern
  - Encapsulate related functionality and create private state, exposing only what you want.
  - Common Use Case:
    - Utilities, Services, & Libraries
- Prototype Pattern
  - Share properties and methods across all instances via prototype inheritance.
  - Common Use Case:
    - When you want to create many objects with shared behavior.

Micro frontend architecture

- Micro Frontends is an approach where a single frontend app is split into smaller, semi-independent apps, each owned by different teams, and integrated together at runtime.
- Extend the microservices idea to the frontend.
  - Microservices = small backend services working together.
  - Micro frontends = small frontend apps working together.
- Core Principles of Micro frontend
  - Different micro frontends can use different dependencies or versions
  - Never share runtime, even if using the same framework
  - Use APIs and events to communicate
  - Each micro frontend can be built and deployed separately

Component-based Architecture

Component-based Architecture

- Component-based architecture is a design approach where the UI is broken down into independent, reusable pieces called components.
- Each component manages its own **structure, logic, and styling**, and can be composed to build complex user interfaces.
- Key Concepts

Concept	Description
Component	A self-contained block of code (JavaScript + JSX + CSS) that renders part of the UI.
Props	Inputs to a component – used to pass data and configuration.
State	Internal data of a component that can change over time and cause re-render.
Composition	Components can include other components – building UIs.

Component-based Architecture

- Benefits of Component-Based Architecture
  - Reusability
    - Components can be reused in multiple places.
  - Maintainability
    - Easier to manage isolated pieces of UI logic.
  - Testability
    - Individual components can be unit-tested independently.
  - Scalability
    - Easy to scale the UI by adding new components.
  - Separation of Concerns
    - Logic, markup, and styling are encapsulated within components.

Creating Reusable Components

- Reusable components are the core of React's power.
- A reusable component is designed to be used in multiple places with different **data or behaviors**, without duplicating code.

Benefit	Explanation
DRY	Don't Repeat Yourself — reuse code instead of duplicating it.
Consistency	UI looks and behaves consistently across the app.
Maintainability	Fix or update in one place = change everywhere.
Scalability	Easier to extend and reuse in larger projects.

State Management in Redux (Redux Toolkit)

- Redux Toolkit (RTK) is the official, recommended way to use Redux.
- It simplifies state management by reducing boilerplate and providing powerful utilities like createSlice and configureStore.
- createSlice
  - Auto-generates reducers and actions
- configureStore
  - Sets up Redux store with good defaults
- Immutability & DevTools
  - Built-in support
- Less Boilerplate
  - Compared to classic Redux

Routing and Navigation

Routing and Navigation

- In React, routing allows you to create multiple views/pages in a single-page application (SPA) using URL paths — without reloading the page.
- React doesn't have built-in routing.
- Use popular library **React Router DOM**, Which is Standard way for routing in React apps.
- Install React Router

```
npm install react-router-dom
```

Handling REST API calls (Axios)

- Axios is a promise-based HTTP client for JavaScript used to interact with REST APIs. It works both in the browser and Node.js environments.
- In a React app, Axios is commonly used to send GET, POST, PUT, DELETE requests to servers.
- Why Use Axios?
  - Easy syntax compared to native fetch()
  - Automatically transforms JSON data
  - Supports request/response interceptors
  - Works with async/await
  - Can set global base URL and headers

Real-time Updates with WebSockets

- WebSockets allow full-duplex communication between the client (React app) and the server — perfect for real-time features like chat apps, live notifications, online presence, stock tickers, etc.
- WebSockets keep a persistent connection open.
- Unlike HTTP (which is request/response), WebSockets allow two-way communication at any time.
- Setting Up WebSocket in React

```
npm install socket.io-client
```

Styling

Styling

- Deep drive knowledge in CSS3 (Grid, Flex etc.)
- CSS Preprocessors (SASS/LESS)
- Bootstrap Integration in React App

Deep drive knowledge in CSS3 (Grid, Flex)

- CSS Flexbox (Flexible Box Layout)
  - Best for 1D layout — aligning items horizontally (row) or vertically (column).
- Flex Container Properties

Property	Description
display: flex	Enables flex context
flex-direction	row (default), column, row-reverse, column-reverse
flex-wrap	Wrap items: nowrap (default), wrap, wrap-reverse
justify-content	Main axis alignment: flex-start, center, space-between, space-around, space-evenly
align-items	Cross axis alignment: stretch, center, flex-start, flex-end, baseline
align-content	Align rows if wrapped (multi-line)

Deep drive knowledge in CSS3 (Grid, Flex)

- CSS Flexbox (Flexible Box Layout)
  - Flex Item Properties

Property	Description
flex	Shorthand for flex-grow, flex-shrink, flex-basis
flex-grow	Item grows to fill space
flex-shrink	Item shrinks if needed
flex-basis	Initial size before growing/shrinking
align-self	Overrides align-items for a single item
order	Reorders items without changing HTML

Deep drive knowledge in CSS3 (Grid, Flex)

- CSS Grid Layout
  - Best for 2D layout — aligning both rows and columns simultaneously.
  - Grid Container Properties

Property	Description
display: grid	Enables grid context
grid-template-columns	Define column sizes: repeat(3, 1fr)
grid-template-rows	Define row sizes
gap or row-gap/column-gap	Adds space between items
grid-template-areas	Name areas for visual layout

Deep drive knowledge in CSS3 (Grid, Flex)

- CSS Grid Layout
  - Grid Item Properties

Property	Description
grid-column	grid-column: 1 / 3 (start / end)
grid-row	Same as above, but for rows
grid-area	Position by area name
justify-self	Align horizontally inside grid cell
align-self	Align vertically inside grid cell

CSS Preprocessors (SASS/LESS)

- A CSS preprocessor is a scripting language that extends standard CSS with features like variables, nesting, mixins, functions, and modular imports — then compiles down to regular CSS.
- The most common CSS preprocessors are:
  - SASS (Syntactically Awesome Style Sheets)
  - LESS (Leaner Style Sheets)

CSS Preprocessors (SASS/LESS)

- Why Use a Preprocessor?

Feature	Benefit
Variables	Reuse colors, font sizes, spacing
Nesting	Structure styles like HTML
Mixins	Reusable code blocks with or without parameters
Functions	Logic like lighten(), darken() on colors
Partials	Split large CSS into modular files (_buttons.scss)
Inheritance	Extend styles from other classes

CSS Preprocessors (SASS/LESS)

- SASS vs LESS — Quick Comparison

Feature	SASS	LESS
Syntax	.scss / .sass	.less
Popularity	High (Bootstrap 5 uses SCSS)	Moderate
Functions	Rich (color ops, math, etc.)	Limited built-ins
Community	Strong, mature	Less active
Recommended?	Yes	Optional

Bootstrap

- Bootstrap is a popular open-source CSS framework for developing responsive and mobile-first websites. It provides:
  - Predefined CSS classes for layout, typography, buttons, forms, etc.
  - Built-in JavaScript plugins (modals, carousels, dropdowns, etc.)
  - A grid system based on Flexbox
  - Responsive design utilities
- There are 2 main approaches to use Bootstrap in a React App
  - Using Bootstrap via CDN
  - Install Bootstrap via NPM

```
npm install bootstrap
```

Build Tools: Webpack

- Webpack is a module bundler. It bundles JS, CSS, images, etc., into optimized static assets. It has been the industry standard for years.
- Key Features:
  - Bundles everything into JS, CSS, HTML, etc.
  - Plugin-rich and highly configurable.
  - Supports Hot Module Replacement (HMR).
  - Slower dev server start time.
  - Mature ecosystem.

Build Tools: Vite

- Vite is a next-generation front-end tool built for lightning-fast development.
- It uses native ES modules in development and Rollup for production bundling.
- Key Features:
  - Instant server start (no bundling during dev).
  - Fast HMR (only reloads changed modules).
  - Uses ES modules and Rollup internally.
  - Supports JS, TS, Vue, React, Svelte, etc.
  - Built-in support for modern frameworks.

Unit Testing: Jest

- Unit Testing is the process of testing individual units of code (typically functions or components) to ensure they work as expected in isolation.
- Jest is a JavaScript Testing Framework developed by Facebook.
- It's widely used for unit testing React apps (but works with any JS/TS code). It supports:
  - Snapshot testing
  - Mocking
  - Code coverage
  - Easy integration with React Testing Library

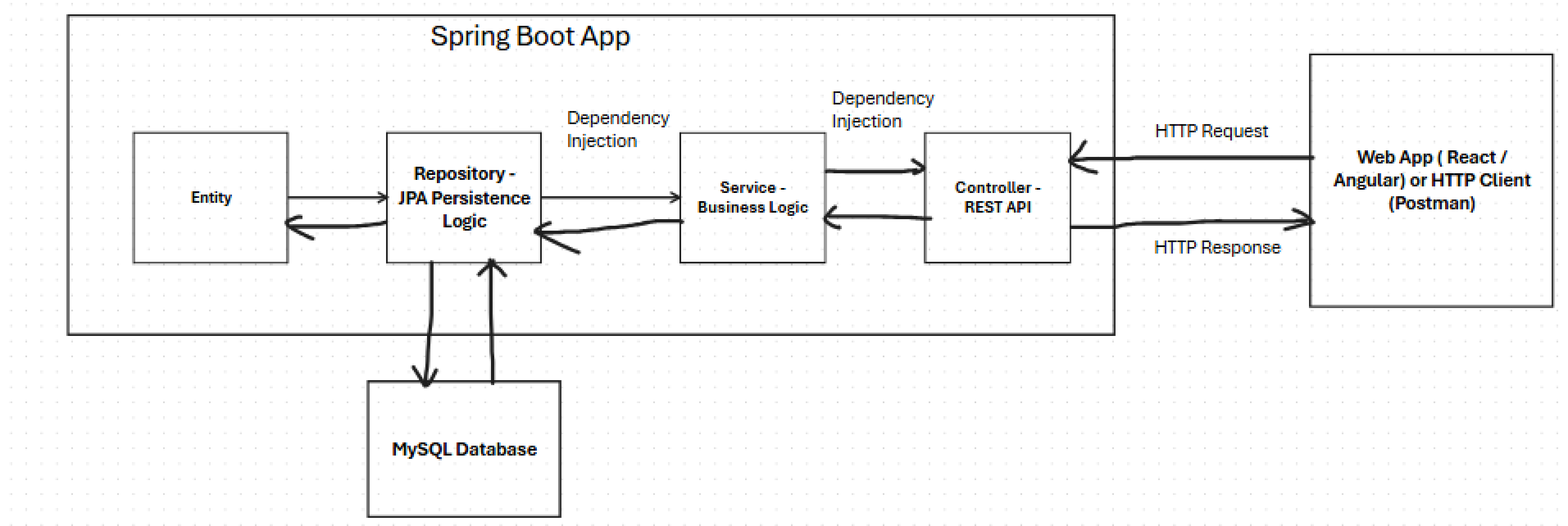
Module 4: Full Stack Integration

Module 4: Full Stack Integration

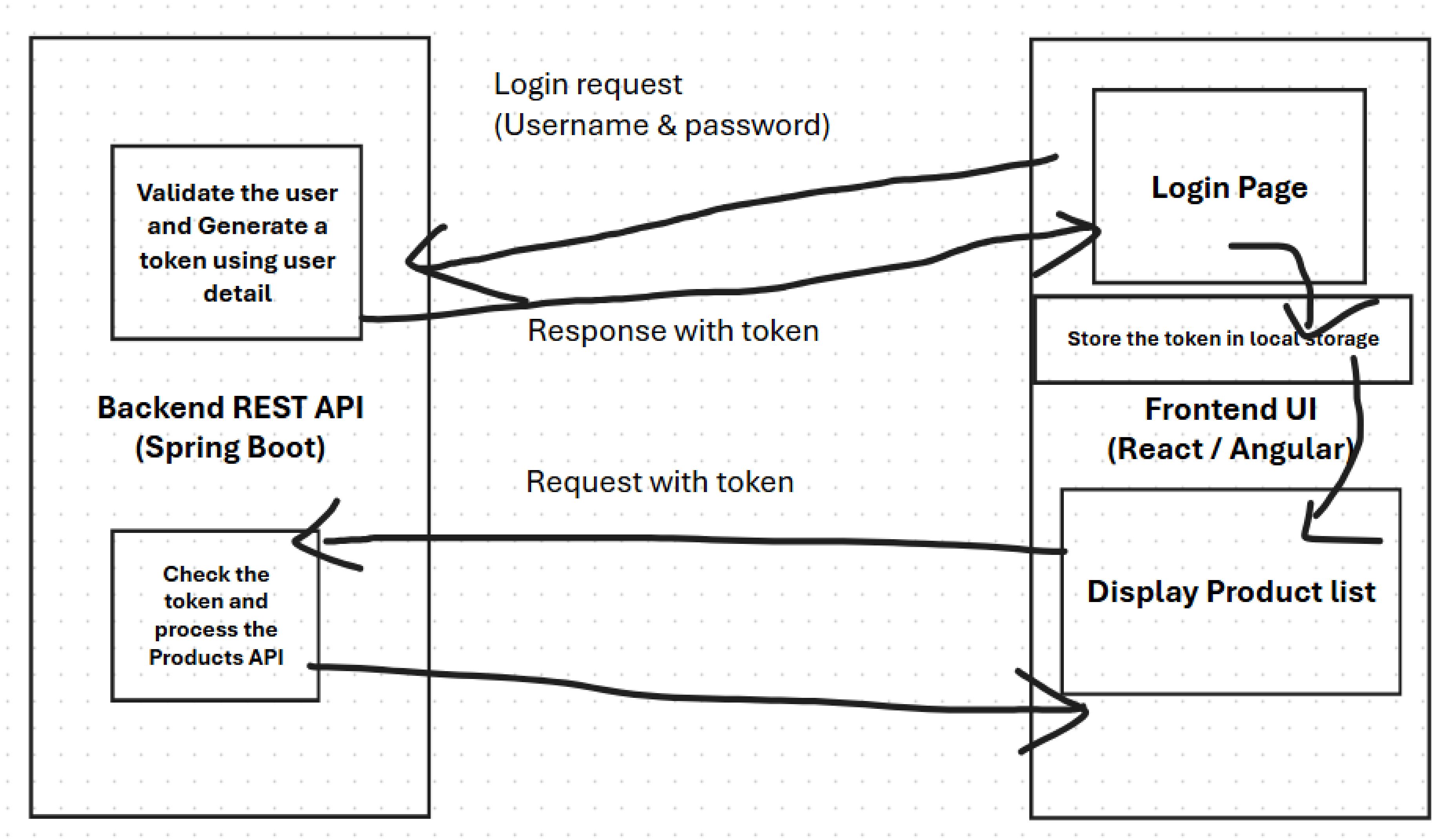
- Connecting Frontend and Backend:
  - API Integration (Frontend consuming Backend APIs)
  - Authentication and Authorization Flow
  - Handling CORS Issues
- Deployment And Tools:
  - Dockerizing Full Stack Applications
  - Kubernetes deployment for Java applications
- Performance Optimization: (Good to Have)
  - Caching
  - Lazy Loading and Pagination
- Testing:
  - API Testing with Postman
  - How to Run Postman Scripts from Newman CLI Commands (Good to Have)

API Integration (Frontend consuming Backend APIs)

- API Integration is a core aspect of building modern web applications where the frontend (UI) communicates with backend services (databases, authentication, business logic) over HTTP.



Authentication and Authorization Flow



Handling CORS Issues

- Handling CORS Issues (Cross-Origin Resource Sharing) is a common challenge when your frontend (React/Angular) calls a backend API hosted on a different origin (domain, port, or protocol).
- CORS is a browser security feature that blocks requests from one origin (e.g., http://localhost:3000) to another (e.g., http://localhost:8080) unless the server explicitly allows it.
- To Fix the Issue, Enable CORS for the frontend origin on the Backend.

Dockerizing Full Stack Applications

- Docker is a Lightweight containerization platform
- Docker Packages app + dependencies into a portable container
- Dockerizing Full Stack Apps ensures consistent environments, Simplifies deployment and Isolates frontend/backend services
- Dockerizing Backend (Spring Boot + MySQL)
  - Add Dockerfile to Spring Boot project
  - Use JAR and expose port
- Dockerizing Frontend (React)
  - Add Dockerfile using nginx
  - Serve static files

Kubernetes deployment for Java applications

- Kubernetes is an open-source container orchestration platform automates deployment, scaling, and management.
- Steps to Deploy Spring Boot App
  - Containerize app using Docker
  - Push image to DockerHub
  - Create Kubernetes deployment.yaml and service.yaml
  - Apply with kubectl apply -f

Performance Optimization: Caching

- Caching improves performance by storing frequently accessed data closer to the user or application, reducing computation time and database load.
- Caching is the process of temporarily storing data in memory so future requests for that data can be served faster.

Benefit	Description
Faster Response Time	Data served from cache is much faster
Reduced DB Load	Avoids repeated DB or API hits
Scalability	Handles more users with the same backend
Cost Efficient	Fewer DB reads and computations

Lazy Loading

- Lazy Loading is a performance optimization technique where resources (like images, components, data, or routes) are loaded only when needed, rather than all at once during the initial load.

Benefit	Description
Faster Initial Load	Only essential data is loaded on page load
Reduced Bandwidth	Unused resources are not fetched immediately
Better User Experience	Feels faster on slow networks/devices
Scalable Apps	Especially useful in large apps/modules

Pagination

- Pagination is the process of dividing a large set of data into smaller chunks (pages), so users can navigate through the content without overwhelming the UI or backend.

Benefit	Description
Improves performance	Only loads data that’s needed at a time
Easier to manage	Reduces memory and API load
Better UX	More user-friendly than dumping all data at once
Enables testing	Makes data operations predictable and modular

API Testing with Postman

- Postman is a powerful GUI-based tool used for developing, testing, and documenting RESTful APIs.
- It allows you to send HTTP requests and view responses quickly and easily.

Feature	Description
Easy to Use	GUI-based interface for sending API requests
Environment Vars	Manage environments for dev, staging, prod easily
Collections	Save grouped requests to share and reuse
Runner	Run multiple test cases in sequence
History	Auto-logs all past requests

How to Run Postman Scripts from Newman CLI Commands

- Newman is a command-line tool developed by the makers of Postman to run Postman collections directly from the terminal.
- It allows automated testing of APIs using Postman collections.
- Install Newman Globally

```
npm install -g newman
```

- Export Your Postman Collection

  In Postman, click on your collection → Export → choose Collection v2.1 format.

- Run with Newman CLI

```
newman run <collection-file>.json
```

Module 5: Microservices Architecture

Module 5: Microservices Architecture

- Introduction to Microservices:
  - Monolith vs Microservices
  - Principles of Microservices Architecture
- Building Microservices: (High Level)
  - Spring Cloud (Netflix OSS)
  - Service Discovery with Eureka
  - API Gateway with Spring Gateway
  - Load Balancing with Ribbon

Module 5: Microservices Architecture

- Inter-Service Communication: REST
- Distributed Systems:
  - Circuit Breakers with Resilience4j
  - Distributed Logging and Monitoring
  - Tracing with Zipkin or Jaeger
- Deploying Microservices with Kubernetes

Introduction to Microservices

Introduction to Microservices

- Microservices Architecture is a design approach where an application is composed of many small, independent services, each focused on a single business capability.
- These services communicate with each other over standard protocols (typically HTTP/REST or messaging queues like RabbitMQ or Kafka)
- Microservices Architecture splits the application into smaller, manageable components, each:
  - Deployed independently
  - Scaled independently
  - Developed and owned by small teams

Monolith vs Microservices

Feature	Monolithic Architecture	Microservices Architecture
Definition	Single unified application with all modules tightly integrated	A set of small, independent services communicating via APIs
Structure	One codebase, one deployable unit	Multiple codebases, independently deployable units
Deployment	Deployed as a single application	Each service deployed independently
Technology Stack	Usually one tech stack	Different services can use different tech stacks
Database	Often a single shared database	Each service has its own database

Monolith vs Microservices

Feature	Monolithic Architecture	Microservices Architecture
Development Team	Suited for small teams	Encourages multiple teams working in parallel
Scalability	Horizontal scaling is at whole-app level	Independent scaling of services
Resilience	A failure in one module can crash the entire app	Failure in one service does not affect others
Maintainability	Harder to maintain as it grows	Easier to maintain smaller, focused services
Testing	Easier integration testing; complex unit testing	Easier unit testing; complex integration and contract testing

Monolith vs Microservices

Feature	Monolithic Architecture	Microservices Architecture
Release Cycle	Slower, coordinated release cycle	Faster, independent deployments
Initial Development	Simple to start and develop	Complex setup with more tooling
DevOps & Infrastructure	Less demanding	Requires strong DevOps practices and orchestration (e.g., Docker, K8s)
Performance	May perform better due to fewer network hops	Slight latency due to network calls between services
Examples	Small web apps, internal tools	E-commerce platforms, streaming services, SaaS platforms

Principles of Microservices Architecture

- Single Responsibility Principle
  - Each microservice should focus on one specific business capability
- Decentralized Data Management
  - Each service owns its own database.
- Independent Deployment
  - Each microservice should be deployable independently without affecting others.
- Smart Endpoints and Dumb Pipes
  - Services contain the logic (smart endpoints), while communication mechanisms are simple (e.g., REST or messaging).

Principles of Microservices Architecture

- Technology Diversity (Polyglot Programming)
  - Teams are free to use different languages and technologies as per service needs.
- Failure Isolation
  - A failure in one microservice should not crash the entire system.
- Infrastructure Automation
  - Use CI/CD pipelines, automated testing, containerization (e.g., Docker), and orchestration (e.g., Kubernetes).
- Scalability
  - Each service should scale independently based on its own load.

Principles of Microservices Architecture

- Security
  - Apply security at the service level.
- Observability
  - Services should be observable through logs, metrics, and tracing.
- Domain-Driven Design (DDD)
  - Structure microservices around business domains, not technical layers.
  - Align services with business boundaries (Bounded Contexts).

Building Microservices

Building Microservices

- Spring Cloud
- Service Discovery with Eureka
- API Gateway with Spring Gateway
- Load Balancing

Spring Cloud

- Spring Cloud is a set of tools and libraries built on top of Spring Boot to help developers build robust, scalable, and resilient cloud-native microservices.
- It provides out-of-the-box solutions for common microservices challenges.
- Spring Cloud simplifies the development of microservices by providing set of tools.

Spring Cloud

- The distributed nature of microservices brings challenges.
- Spring helps you mitigate these with several ready-to-run cloud patterns.
- Spring Cloud can help with service discovery, load-balancing, circuit-breaking, distributed tracing, and monitoring.
- It can even act as an API gateway.

Service Discovery with Eureka

- In a microservices architecture, each service may have multiple instances (e.g., running on different ports or machines).
- These instances can scale dynamically, start, or shut down at any time.
- Service Discovery is a mechanism that helps services find and communicate with each other without hardcoding network addresses.

Service Discovery with Eureka

- Eureka is a Service Discovery server provided by Spring Cloud Netflix. It acts as a registry where:
  - Services register themselves on startup.
  - Other services query the registry to find addresses of available service instances.
- Eureka Server
  - The central registry where all microservices register.
- Eureka Client
  - Each microservice registers itself and discovers other services.

API Gateway with Spring Gateway

- An API Gateway is a single entry point for all client requests to your microservices architecture.
- It acts as a reverse proxy, routing requests to backend services.
- Spring Cloud Gateway is the modern API Gateway solution from the Spring ecosystem.
- It's built on Spring WebFlux (reactive), and replaces the older Zuul 1 from Netflix OSS.

Load Balancing with Ribbon

- Load Balancing is the process of distributing client requests across multiple instances of a service to:
  - Maximize availability and fault tolerance
  - Improve scalability and performance
  - Avoid overloading any single instance
- In Spring Cloud, Load Balancing allows your microservices to communicate efficiently when multiple instances of a target service exist

Inter-Service Communication: REST

Inter-Service Communication: REST

- Inter-Service Communication refers to how microservices talk to each other to fulfill business operations.
- In a microservices architecture, services are distributed and often need to communicate to:
  - Share data
  - Trigger events
  - Coordinate workflows

Inter-Service Communication: REST

- REST (Representational State Transfer)
  - REST is the most widely used method for service-to-service communication using standard HTTP verbs.
- Pros:
  - Simple and widely adopted
  - Human-readable (JSON/XML)
  - Works over standard HTTP/1.1
  - Easy testing with tools like Postman or curl
- Cons:
  - Slower due to text format and no compression
  - Not suitable for real-time or streaming
  - Limited support for contract enforcement

Distributed Systems

Distributed Systems

- Distributed Systems:
  - Circuit Breakers with Resilience4j/Hystrix
  - Distributed Logging and Monitoring
  - Tracing with Zipkin or Jaeger
- Deploying Microservices with Kubernetes

Circuit Breakers with Resilience4j

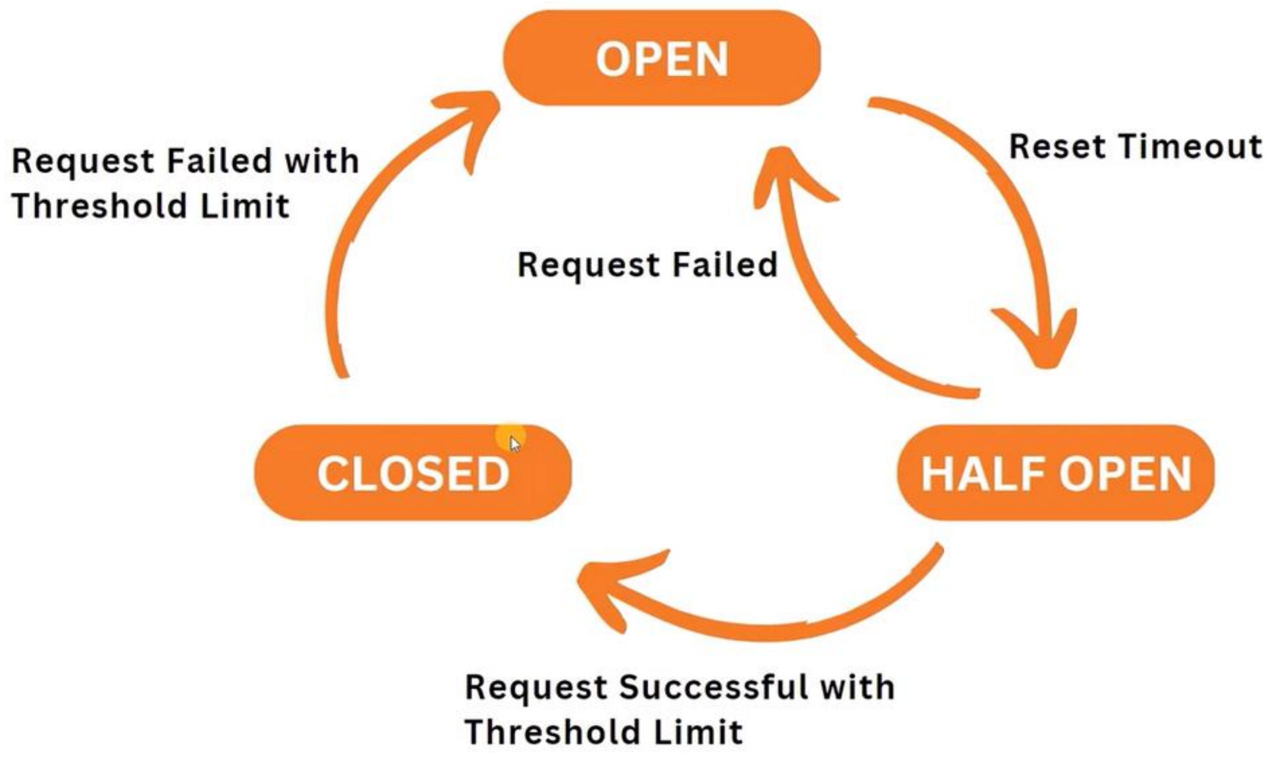
- A Circuit Breaker is a resilience pattern used to prevent cascading failures in a distributed system.
- It acts like an electrical circuit breaker: if a service starts failing (e.g., due to network issues or being down), the circuit breaker opens and stops further calls to that service for a period of time.
- In microservices, one failing service can slow down or crash others if they keep waiting for responses.
- Spring Boot uses Resilience4j as a preferred circuit breaker implementation

Circuit Breakers with Resilience4j

- Circuit breakers help by
  - Fail Fast
    - Quickly fail when the service is down instead of waiting
  - Protect Resources
    - Save threads/CPU by avoiding unnecessary retries
  - Auto Recovery
    - Retry the service after a cooldown time
  - Improve Stability
    - Avoid complete system collapse due to one faulty service

Circuit Breakers with Resilience4j

- Circuit Breaker States
- Closed
  - All requests go through normally
- Open
  - All requests fail fast (no calls made)
- Half-Open
  - A few test requests are allowed; if successful, return to closed


OPEN
Reset Timeout
Request Failed with Threshold Limit
Request Failed
CLOSED
HALF OPEN
Request Successful with Threshold Limit

Distributed Logging and Monitoring

- In a microservices architecture, multiple services run independently, often on different servers, containers, or cloud environments.
- To observe, debug, and maintain such a distributed system, we need centralized and correlated visibility.
- Distributed Logging
  - Distributed Logging means collecting and centralizing logs from all microservices in one place so you can:
    - Debug issues across service boundaries
    - Trace request flow between services
    - Correlate logs with timestamps, request IDs, or trace IDs

Distributed Logging and Monitoring

- Distributed Logging Common Tools

Tool	Purpose
Logback/Log4j	Local logging in Spring Boot
Filebeat/Fluentd	Log shipper (collect logs)
Logstash	Log processing pipeline
Elasticsearch	Log storage and search engine
Kibana	Log visualization and analysis UI

Distributed Logging and Monitoring

- Distributed Monitoring
  - Monitoring means continuously collecting metrics and health information from services to detect:
    - High latency
    - Increased error rate
    - Resource usage (CPU, memory)
    - Service downtime
  - Common Monitoring Tools

Tool	Role
Spring Boot Actuator	Exposes health, metrics, env
Micrometer	Metrics facade for Prometheus, etc.
Prometheus	Pull-based time-series metrics DB
Grafana	Visualization for metrics

Tracing with Zipkin

- Tracing is a technique used in microservices to track the flow of a single request as it travels across multiple services.
- This is known as distributed tracing, and it's essential for debugging, monitoring latency, and understanding the lifecycle of a request in complex systems.
- Spring Cloud Sleuth
  - Adds trace IDs to logs and headers
- Zipkin
  - Collects and visualizes traces

Deploying Microservices with Kubernetes

- Kubernetes (K8s) is the de facto standard for deploying, scaling, and managing containerized microservices.
- It allows you to manage each microservice independently and ensures high availability, auto-scaling, self-healing, and service discovery.

Happy Learning :)