GSDC
Global Skill Development Council

JAVA FULL STACK DEVELOPMENT

Frameworks

www.gsdcouncil.org

Introduction to Java Full Stack Development

A Java Full Stack Developer works across every layer of a modern application — from what users see in their browser to how data is stored and retrieved from databases. Mastering the full stack means understanding not just individual technologies, but how each layer communicates, integrates, and scales together in a production environment. Frameworks are the backbone of this approach, enabling teams to build faster, maintain consistency, and deliver reliable software at scale.

Frameworks reduce repetitive boilerplate code, enforce design patterns, and provide battle-tested solutions to common problems. Rather than reinventing the wheel, developers can focus on delivering business value. The Java ecosystem offers an exceptionally rich set of frameworks covering every aspect of application development, from responsive user interfaces to container orchestration.

User Interface

Angular, React, and Vue.js power modern, responsive front-end experiences.

Back-End Logic

Spring Framework and Spring Boot handle business logic and application orchestration.

Persistence Layer

Hibernate and JPA bridge Java objects to relational and non-relational databases.

Security

Spring Security, JWT, and OAuth2 protect application endpoints and user data.

Layer	Common Frameworks & Tools
Front-End	Angular, React, Vue.js
Styling	Bootstrap, Tailwind CSS
Back-End	Spring Framework, Spring Boot
Persistence	Hibernate, JPA
APIs	Spring MVC, Spring REST
Security	Spring Security
Database	MySQL, PostgreSQL, MongoDB
Build Tools	Maven, Gradle
Testing	JUnit, Mockito
DevOps	Docker, Kubernetes, Jenkins

Front-End Frameworks: Angular

Front-end frameworks provide the structure, tooling, and conventions needed to build rich, interactive, and responsive user interfaces that run in the browser. Without a front-end framework, developers would have to manually manage DOM manipulation, state management, and routing — tasks that quickly become unmanageable at enterprise scale. Angular stands out as the go-to choice for large teams building complex, long-lived single-page applications.

Angular is a TypeScript-based framework developed and maintained by Google. It follows a strictly opinionated, component-based architecture that enforces consistency across large codebases. Its built-in support for dependency injection, reactive forms, and an HTTP client makes it a self-contained platform rather than just a UI library. This comprehensive nature makes Angular particularly well-suited for enterprise environments where maintainability and testability are critical concerns.

Angular Key Features

- Component-based architecture
- Two-way data binding
- Built-in dependency injection
- Powerful routing support
- Reactive and template-driven forms
- Built-in HTTP client services
- Strong TypeScript integration

Angular Architecture

Component	Purpose
Components	UI building blocks
Services	Business logic
Modules	Application organization
Routing	Navigation management
Directives	DOM manipulation

Angular offers strong enterprise adoption, rich ecosystem support, and built-in testing utilities — ideal for large-scale SPAs.

Front-End Frameworks: React & Vue.js

While Angular provides a complete platform, React and Vue.js offer more flexibility and a lighter footprint, making them popular choices for teams that want to compose their own toolchains. React, developed by Meta, introduced the concept of a Virtual DOM and popularized the component-based model that now dominates front-end development. Vue.js, created by Evan You, blends ideas from Angular and React into a progressive framework that is celebrated for its gentle learning curve and powerful capabilities.

React

- Virtual DOM for fast rendering
- Reusable component model
- Massive community and ecosystem
- Flexible — pairs with Redux, React Router
- Ideal for dashboards and SPAs

Vue.js

- Progressive and incrementally adoptable
- Easy learning curve for new developers
- Lightweight with fast performance
- Flexible integration with existing projects
- Single-file components (.vue)

React is particularly well-suited for projects requiring high interactivity and complex state management. Its unidirectional data flow encourages predictable application behavior, which scales well as applications grow. Vue.js, on the other hand, is a favourite for teams transitioning from jQuery-era applications or for projects where developer velocity and low onboarding friction are top priorities.

Front-End Styling Frameworks

Styling frameworks accelerate UI development by providing pre-built design systems, component libraries, and utility classes that ensure visual consistency and responsive behavior across devices. Choosing the right styling framework often comes down to project requirements: teams that need rapid prototyping with ready-made components typically lean toward Bootstrap, while teams that prefer full design control and performance-optimized output often choose Tailwind CSS.

Bootstrap

Bootstrap remains one of the most widely adopted CSS frameworks in the world. It provides a responsive 12-column grid system, a comprehensive set of UI components, and built-in JavaScript plugins. Its extensive documentation and large community make it accessible to developers at all experience levels.

Grid System

12-column responsive layout

Components

Buttons, modals, navbars, cards

Compatibility

Cross-browser support

Tailwind CSS

Tailwind CSS takes a utility-first approach, providing low-level utility classes that developers compose directly in HTML markup. This results in highly customizable designs without writing custom CSS files. Tailwind's JIT (Just-In-Time) compiler produces minimal CSS bundles, improving load performance significantly.

- Utility-first design philosophy
- Rapid UI development workflow
- Fully customizable design tokens
- Minimal CSS output via JIT compiler
- Modern development approach

Both Bootstrap and Tailwind CSS integrate seamlessly with Angular, React, and Vue.js projects through npm packages.

Java Back-End Frameworks: Spring Framework

The Spring Framework is the cornerstone of enterprise Java development. First released in 2003, Spring was designed as a lightweight alternative to Java EE, offering a modular, POJO-based programming model that eliminated the complexity of heavy container-managed components. Today, Spring has evolved into a complete ecosystem that addresses virtually every concern of modern application development.

Spring's power lies in its modular architecture. Developers can choose only the modules they need, keeping applications lean while having the option to add capabilities incrementally. The framework's Inversion of Control container manages the lifecycle of application components, promoting loose coupling and making code significantly easier to test and maintain. Spring's massive adoption in the industry means a wealth of learning resources, third-party integrations, and community support.

Module	Purpose
Spring Core	Dependency Injection and IoC container
Spring MVC	Web application development (Model-View-Controller)
Spring AOP	Cross-cutting concerns (logging, transactions)
Spring Data	Simplified database operations and repository pattern
Spring Security	Authentication and Authorization
Spring Boot	Auto-configuration and rapid application development

Modular Architecture

Use only what you need — Spring's modules are independently adoptable, keeping your application footprint minimal.

Enterprise Scalability

Proven at massive scale across banking, retail, healthcare, and technology industries worldwide.

Strong Ecosystem

Deep integrations with Hibernate, Kafka, Redis, cloud providers, and hundreds of third-party libraries.

Dependency Injection & Inversion of Control

Dependency Injection (DI) and Inversion of Control (IoC) are two fundamental principles that Spring is built upon. Understanding these concepts is essential for any Java developer working with the Spring ecosystem. Together, they transform how objects are created and wired together, replacing manual instantiation and tight coupling with a managed, flexible, and highly testable component model.

Dependency Injection (DI)

DI is a design pattern that removes the responsibility of creating dependencies from a class itself. Instead of a class instantiating the objects it needs, those objects (dependencies) are provided — or "injected" — from the outside. This reduces tight coupling between components, making the codebase more modular and easier to test with mock implementations.

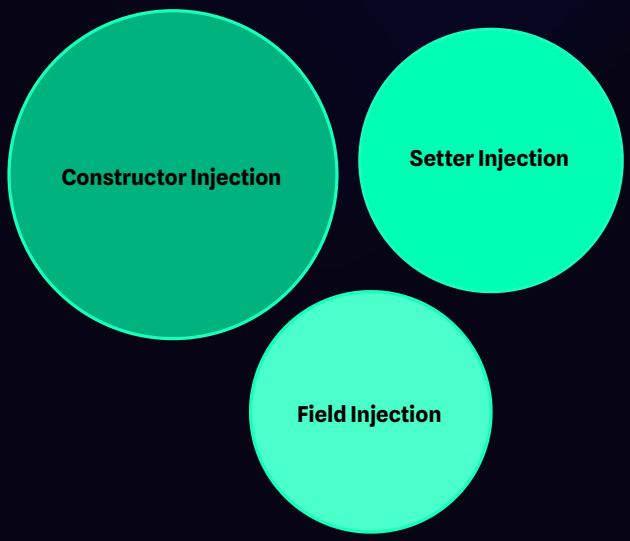
Types of Injection

- **Constructor Injection** — Dependencies provided via constructor parameters (recommended)
- **Setter Injection** — Dependencies set via setter methods after construction
- **Field Injection** — Dependencies injected directly into fields using `@Autowired`

Inversion of Control (IoC)

IoC refers to the principle where the framework, not the developer, controls object creation and lifecycle management. The Spring IoC container reads configuration metadata (annotations or XML) and automatically instantiates, configures, and assembles the objects (called beans) your application needs. This inversion of responsibility means developers focus on business logic while Spring handles infrastructure concerns.

> Constructor injection is the strongly recommended approach as it makes dependencies explicit, supports immutability, and simplifies unit testing without requiring a Spring context.



Spring's IoC container scans the application for annotated classes, resolves the dependency graph, and injects the appropriate implementations at runtime. This approach enables powerful patterns like coding to interfaces, making it trivial to swap implementations — for example, replacing a real database repository with an in-memory mock during testing.

Spring Boot Framework

Spring Boot represents a paradigm shift in how Spring applications are built and deployed. Before Spring Boot, configuring a Spring application required significant XML or Java configuration — setting up data sources, transaction managers, MVC dispatchers, and embedded servers was time-consuming and error-prone. Spring Boot eliminates this friction through its auto-configuration mechanism, which intelligently configures your application based on the dependencies present on the classpath.

Spring Boot's opinionated defaults mean developers can go from project creation to a running application in minutes. The embedded server support (Tomcat, Jetty, Undertow) means applications are packaged as self-contained executable JARs, dramatically simplifying deployment. Spring Boot Actuator provides production-ready monitoring endpoints out of the box, including health checks, metrics, and environment information.

1

Controller Layer

Handles HTTP requests and routes to appropriate service methods

2

Service Layer

Contains business logic and orchestrates operations

3

Repository Layer

Manages all database interactions via Spring Data

Common Spring Boot Annotations

Annotation	Purpose
`@SpringBootApplication`	Marks the main application entry point; combines @Configuration, @EnableAutoConfiguration, @ComponentScan
`@RestController`	Marks a class as a REST API controller, combining @Controller and @ResponseBody
`@Service`	Marks a class as a business logic service component
`@Repository`	Marks a class as a data access object (DAO) with exception translation
`@Autowired`	Triggers automatic dependency injection by Spring's IoC container

Rapid Development

Auto-configuration and starters eliminate boilerplate setup code

Easy Deployment

Self-contained executable JARs with embedded servers

Cloud-Ready

First-class support for microservices and cloud-native patterns

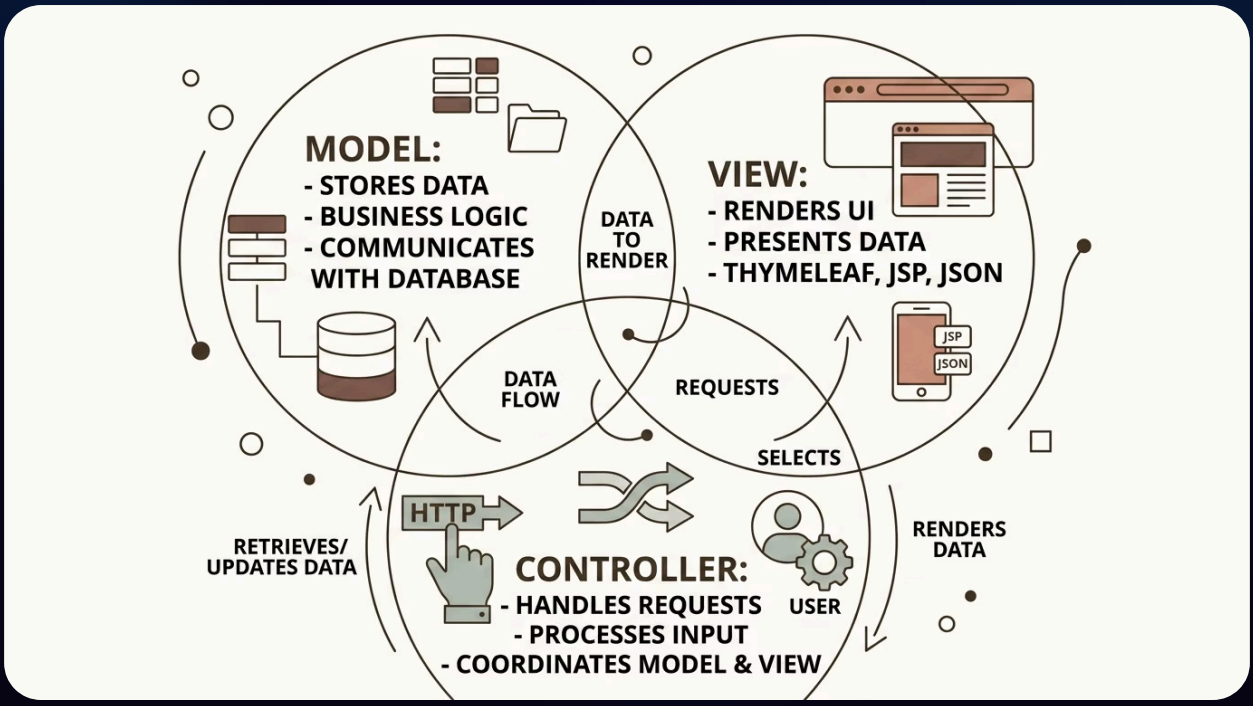
Production Features

Built-in Actuator for health checks and metrics monitoring

Spring MVC Framework

Spring MVC (Model-View-Controller) is the web framework module within the Spring ecosystem, providing a structured approach to handling HTTP requests in web applications. It implements the classic MVC design pattern, cleanly separating application concerns into three distinct roles: the Model (data and business logic), the View (user interface presentation), and the Controller (request handling and routing logic). This separation of concerns makes applications significantly easier to develop, test, and maintain over time.

In a typical Spring MVC application, the `DispatcherServlet` acts as the front controller, intercepting all incoming requests and delegating them to the appropriate controller based on URL mappings. Spring MVC supports multiple view technologies — from traditional JSP and Thymeleaf templates to JSON responses for REST APIs — making it versatile enough for both traditional server-rendered web applications and modern API-first architectures.



Separation of Concerns

Each layer has a clearly defined responsibility, preventing logic from bleeding across boundaries and making code easier to reason about and test independently.

Easy Maintenance

Changes to the view layer do not affect business logic, and database changes are isolated to the model layer — reducing the risk of unintended side effects.

Better Scalability

The decoupled architecture allows individual layers to be scaled, refactored, or replaced without rewriting the entire application.

Persistence Frameworks: Hibernate & ORM

Persistence frameworks bridge the gap between the object-oriented world of Java and the relational world of databases. Without an ORM (Object-Relational Mapping) framework, developers must write verbose JDBC code to map result sets to Java objects, handle transactions manually, and manage connection pools. Hibernate, the most widely adopted ORM framework in the Java ecosystem, automates all of these concerns while providing a powerful query language and sophisticated caching capabilities.

Hibernate maps Java classes (entities) to database tables using annotations, eliminating the need for most handwritten SQL. It manages the entity lifecycle — from creation to persistence to deletion — transparently within a session. Hibernate's first and second-level caching mechanisms dramatically reduce database round-trips for frequently accessed data, improving application performance without requiring manual cache management code.

Hibernate Annotations

Annotation	Purpose
`@Entity`	Marks class as a database entity
`@Table`	Maps entity to a specific table
`@Id`	Designates the primary key field
`@Column`	Maps field to a specific column
`@GeneratedValue`	Auto-generates primary key values

Hibernate Relationships

- **@OneToOne** — User → Profile
- **@OneToMany** — Customer → Orders
- **@ManyToOne** — Orders → Customer
- **@ManyToMany** — Students ↔️ Courses

Key Benefits

- Reduces manual SQL coding significantly
- Improves database portability
- Supports first and second-level caching
- Simplifies complex join operations

JPA: Java Persistence API

JPA (Java Persistence API) is a specification — a set of interfaces and annotations — that standardizes how Java applications interact with relational databases using ORM. Rather than tying applications to a specific ORM implementation, JPA defines a common contract that multiple providers (Hibernate, EclipseLink, OpenJPA) can implement. This vendor independence is one of JPA's most significant advantages: applications written against the JPA API can switch ORM providers with minimal code changes.

In practice, Hibernate is the most popular JPA implementation and is the default provider in Spring Boot projects. JPA's EntityManager, JPQL (Java Persistence Query Language), and criteria API provide a powerful and standardized way to query and manage entities. Spring Data JPA further simplifies database access by automatically generating repository implementations from method names, reducing the need for even JPQL queries in many common scenarios.

Vendor Independence

Switch between Hibernate, EclipseLink, or other providers without rewriting business logic or query code.

Standardized Persistence

JPA provides a uniform API across all Java EE and Jakarta EE-compliant containers and standalone applications.

Simplified Operations

Spring Data JPA generates repository implementations from method names, reducing boilerplate to near zero.

Spring Data JPA's method naming convention is remarkably powerful: a method named `findByLastNameAndEmail(String lastName, String email)` automatically generates the correct SQL query without any implementation code.

API Development with Spring Boot REST

RESTful APIs have become the universal language of modern software systems. They enable front-end applications, mobile clients, third-party services, and microservices to communicate over HTTP using a stateless, resource-oriented architecture. Spring Boot provides first-class support for building REST APIs through its `@RestController` annotation model, making it straightforward to expose well-structured, documented, and secure API endpoints.

A well-designed REST API follows a set of conventions that make it predictable and easy to consume. Resources should be identified by meaningful, hierarchical URLs. HTTP methods should be used semantically — GET for retrieval, POST for creation, PUT/PATCH for updates, DELETE for removal. Responses should include appropriate HTTP status codes so clients can handle success and error conditions correctly without parsing response bodies.

HTTP Methods

Method	Purpose
GET	Retrieve a resource or collection
POST	Create a new resource
PUT	Replace an existing resource
PATCH	Partially update a resource
DELETE	Remove a resource

Common HTTP Status Codes

Code	Meaning
200	OK — Request succeeded
201	Created — Resource created successfully
400	Bad Request — Invalid input data
401	Unauthorized — Authentication required
404	Not Found — Resource does not exist
500	Internal Server Error — Unexpected failure

REST API Best Practices

Use Meaningful Endpoints

Design URLs around resources (`/api/customers/{id}`), not actions. Avoid verbs in endpoints.

Version Your APIs

Include versioning in the URL (`/api/v1/`) or headers to maintain backward compatibility as APIs evolve.

Secure All Endpoints

Implement authentication and authorization on every endpoint. Never expose sensitive data without proper access controls.

Validate Inputs

Use Spring's `@Valid` annotation and Bean Validation to reject malformed requests at the API boundary.

Security Frameworks: Spring Security

Security is not optional in modern application development — it is a fundamental requirement that must be designed into every layer of an application from day one. Spring Security is the de facto standard for securing Spring-based applications, providing comprehensive authentication, authorization, and protection against common web vulnerabilities. It integrates seamlessly with Spring Boot and Spring MVC, applying security concerns as a cross-cutting aspect without polluting business logic code.

Spring Security works through a chain of security filters that intercept every incoming HTTP request before it reaches your application code. Each filter handles a specific security concern: authentication, CSRF protection, session management, or header security. This filter chain architecture is highly configurable, allowing teams to tailor security behavior precisely to their application's requirements.

1 — Form Authentication

Traditional username/password login forms with session-based state management

2 — Basic Authentication

HTTP Basic Auth — credentials encoded in request headers, suitable for internal APIs

3 — JWT Authentication

Stateless token-based security — ideal for REST APIs and microservices architectures

4 — OAuth2

Third-party authentication via providers like Google, GitHub, or enterprise identity servers

Security Best Practices

Encrypt Passwords

Always use BCrypt or Argon2 hashing — never store plain text passwords.

Role-Based Access

Restrict resources to users with appropriate roles using `@PreAuthorize`.

Use HTTPS

Enforce HTTPS in production to protect data in transit from interception.

Validate Inputs

Sanitize and validate all user inputs to prevent injection attacks.

Build Automation: Maven & Gradle

Build automation tools are indispensable in modern Java development. They manage project dependencies, compile source code, run tests, and package applications into deployable artifacts — all with a single command. Without build tools, managing dozens of transitive dependencies and ensuring consistent builds across developer machines and CI servers would be an enormous manual burden. Maven and Gradle are the two dominant build automation tools in the Java ecosystem.

Maven

Maven uses a declarative XML-based configuration model centered around the pom.xml file. Its convention-over-configuration philosophy means that projects following standard directory layouts require minimal configuration. Maven's Central Repository hosts hundreds of thousands of open-source libraries, making dependency management straightforward.

Maven Build Lifecycle

- **validate** — Validates project structure
- **compile** — Compiles source code
- **test** — Runs unit tests
- **package** — Creates JAR/WAR artifact
- **install** — Installs to local repository
- **deploy** — Deploys to remote repository

Gradle

Gradle takes a programmatic approach, using Groovy or Kotlin DSL build scripts that offer far greater flexibility than Maven's XML configuration. Gradle's incremental build system and build cache significantly reduce build times for large projects, making it the preferred choice for Android development and increasingly for large Java microservices projects.

Maven vs Gradle Comparison

Feature	Maven	Gradle
Configuration	XML (pom.xml)	Groovy/Kotlin DSL
Performance	Good	Faster (incremental)
Learning Curve	Easier	Moderate
Flexibility	Convention-based	Highly flexible

Testing Frameworks: JUnit & Mockito

Automated testing is one of the most important practices in professional software development. A comprehensive test suite enables developers to refactor code with confidence, catch regressions early, and document expected behavior through executable specifications. Java's testing ecosystem is mature and powerful, anchored by JUnit for test structure and Mockito for dependency mocking.

JUnit 5

JUnit is the foundation of Java testing. JUnit 5 (Jupiter) introduced a more flexible and extensible architecture, powerful parameterized tests, and better integration with IDEs and build tools. Its annotation model makes test lifecycle management clean and explicit.

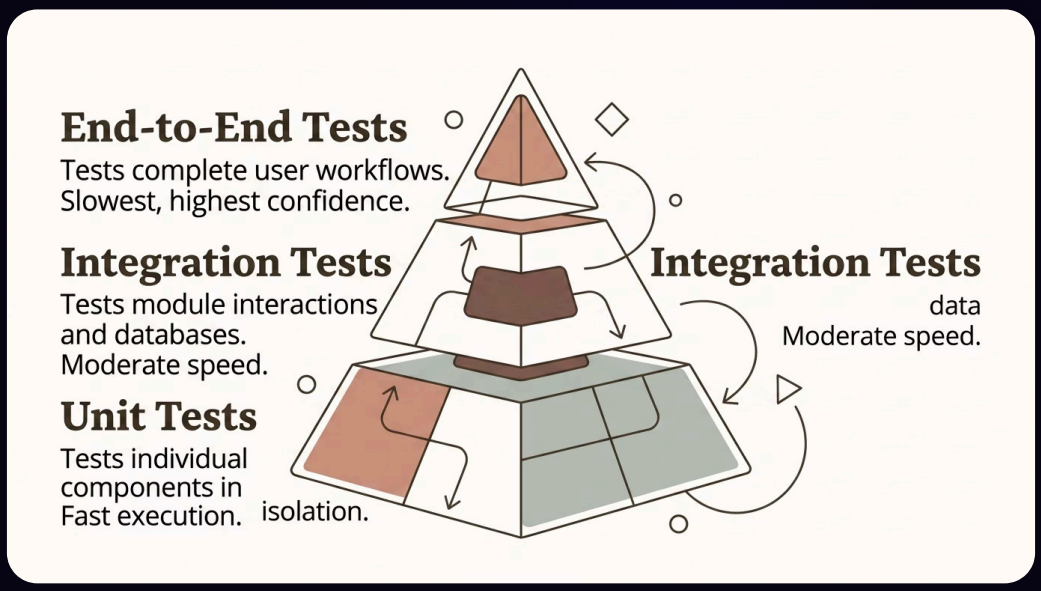
Annotation	Purpose
`@Test`	Marks a test method
`@BeforeEach`	Setup before each test
`@AfterEach`	Cleanup after each test
`@BeforeAll`	One-time setup for class
`@ParameterizedTest`	Data-driven test cases

Mockito

Mockito allows developers to create mock objects that simulate the behavior of real dependencies. This enables true unit testing — testing a single class in complete isolation from its collaborators. Mocks can be programmed to return specific values, throw exceptions, or verify that certain methods were called during a test.

- Isolate the class under test from external dependencies
- Simulate error conditions and edge cases
- Verify method interactions with `verify()`
- Stub return values with `when().thenReturn()`

The Testing Pyramid



The testing pyramid is a guiding principle: invest heavily in fast, focused unit tests at the base, supplement with integration tests that verify module boundaries, and use a small number of end-to-end tests to validate critical user journeys. This distribution maximizes confidence while keeping test suite execution time manageable.

Microservices Frameworks: Spring Boot & Spring Cloud

Microservices architecture decomposes large monolithic applications into a collection of small, independently deployable services that communicate over well-defined APIs. Each service owns its data, can be deployed and scaled independently, and is developed by a small, focused team. Spring Boot and Spring Cloud together form the most widely adopted microservices stack in the Java ecosystem, providing solutions for every infrastructure concern that arises when running distributed systems.

The challenges of microservices — service discovery, distributed configuration, circuit breaking, load balancing, and distributed tracing — are elegantly addressed by Spring Cloud components. Rather than solving these infrastructure problems from scratch, teams leverage battle-tested libraries that integrate seamlessly with the Spring ecosystem they already know.

Config Server

Centralized configuration management across all services. Eliminates environment-specific configuration files from each service.

Eureka Discovery

Service registry that allows services to find and communicate with each other by name rather than hardcoded URLs.

API Gateway

Single entry point for all client requests — handles routing, authentication, rate limiting, and request transformation.

OpenFeign

Declarative HTTP client that simplifies service-to-service communication with interface-based definitions.

Microservices with Spring Cloud enable independent deployment, fault isolation, and technology flexibility — each service can use its own database and be updated without affecting others.

Containerization: Docker

Docker revolutionized application deployment by solving one of the most persistent problems in software development: the dreaded "it works on my machine" scenario. By packaging an application along with all of its dependencies, configuration, and runtime environment into a portable container image, Docker guarantees that software behaves identically in development, testing, staging, and production environments. This consistency dramatically reduces deployment failures and environment-related debugging time.

A Docker container is a lightweight, isolated process that shares the host operating system's kernel but has its own filesystem, network, and process space. This makes containers far more resource-efficient than virtual machines while providing strong isolation. Container images are built from Dockerfiles — declarative text files that specify the base image, application dependencies, and startup commands.

Docker Components

Component	Purpose
Dockerfile	Build instructions for an image
Image	Immutable application template
Container	Running instance of an image
Registry	Storage for distributing images
Compose	Multi-container orchestration

Key Benefits

- **Consistency**

  Identical behavior across all environments from dev to production

- **Portability**

  Run anywhere — laptop, cloud, or on-premise server

- **Faster Deployment**

  Containers start in milliseconds, enabling rapid scaling

- **Isolation**

  Services run independently without interfering with each other

Container Orchestration: Kubernetes

While Docker solves the problem of packaging and running containers, Kubernetes (K8s) solves the problem of managing containers at scale. When an application consists of dozens or hundreds of containerized microservices, manually managing deployment, scaling, networking, and failure recovery becomes impossibly complex. Kubernetes automates these operational concerns, allowing teams to declare the desired state of their application and trust Kubernetes to continuously reconcile the actual state toward that goal.

Kubernetes' self-healing capabilities are particularly powerful: if a container crashes, K8s automatically restarts it. If a node fails, workloads are rescheduled to healthy nodes. Rolling deployments enable zero-downtime updates by gradually replacing old container instances with new ones. Horizontal pod autoscaling automatically increases or decreases the number of running instances based on CPU utilization or custom metrics.

Pod

The smallest deployable unit in Kubernetes — wraps one or more containers that share network and storage.

Node

A worker machine (virtual or physical) that hosts pods and provides compute, memory, and storage resources.

Cluster

A collection of nodes managed by the Kubernetes control plane, forming the complete compute environment.

Deployment

Declarative application management — defines desired replicas, update strategy, and rollback behavior.

0s

Zero Downtime

Rolling updates enable continuous deployment without service interruption

Auto

Self-Healing

Automatic restart and rescheduling of failed containers and nodes

HPA

Auto-Scaling

Horizontal Pod Autoscaler adjusts replicas based on real-time metrics

DevOps Integration: Jenkins & GitHub Actions

Continuous Integration and Continuous Deployment (CI/CD) pipelines are the backbone of modern DevOps practices. They automate the journey from a developer committing code to that code being tested, built, and deployed to production — often in minutes. This automation eliminates manual deployment steps, reduces human error, and enables teams to ship software more frequently and with greater confidence. Jenkins and GitHub Actions are two of the most widely used CI/CD platforms in the Java ecosystem.

Jenkins

Jenkins is an open-source automation server with a vast plugin ecosystem. Its declarative pipeline syntax (Jenkinsfile) allows teams to define their entire CI/CD workflow as code, version-controlled alongside the application. Jenkins supports complex build topologies including distributed builds, parallel stages, and conditional deployments.

Pipeline Stages

1. **Build**
   Compile and package the application with Maven or Gradle
2. **Test**
   Run unit, integration, and code quality checks
3. **Package**
   Build Docker image and push to container registry
4. **Deploy**
   Deploy to Kubernetes or cloud environment

GitHub Actions

GitHub Actions brings CI/CD directly into the GitHub ecosystem, triggered by repository events like pushes, pull requests, or scheduled cron jobs. Workflows are defined in YAML files within the repository, and a rich marketplace of pre-built actions eliminates the need to write complex scripts for common tasks like Docker builds, Maven builds, and Kubernetes deployments.

> GitHub Actions is increasingly preferred for new projects due to its tight GitHub integration, zero infrastructure overhead, and an extensive marketplace of community-contributed actions.

- YAML-based workflow definitions
- Triggered by repository events
- Extensive marketplace of pre-built actions
- Built-in secrets management
- Matrix builds for multiple Java versions

Logging Frameworks: Log4j & SLF4J

Logging is one of the most critical operational concerns for any production application. Without comprehensive, well-structured logs, diagnosing production issues becomes a time-consuming exercise in guesswork. Java's logging ecosystem provides powerful frameworks for capturing, filtering, formatting, and routing log output to various destinations — from the console during development to centralized log aggregation platforms in production.

Log4j (and its successor Log4j 2) is a highly configurable logging framework that supports asynchronous logging, multiple appenders (file, database, messaging systems), and flexible output formats. The SLF4J facade provides a common API that decouples application code from the underlying logging implementation, making it easy to switch between Log4j, Logback, or java.util.logging without changing application code. Spring Boot uses SLF4J with Logback as its default logging stack.

Log Levels Explained

Level	When to Use
TRACE	Very detailed diagnostic information
DEBUG	Troubleshooting and development information
INFO	General operational information
WARN	Potential issues that don't stop execution
ERROR	Failures that need immediate attention
FATAL	Critical failures causing application shutdown

Logging Best Practices

- Use structured logging (JSON) for machine-parseable output
- Include correlation IDs for tracing requests across microservices
- Never log sensitive data such as passwords or PII
- Use appropriate log levels to avoid noise in production
- Configure log rotation to manage disk space
- Ship logs to centralized platforms (ELK Stack, Splunk)

⚠ Avoid logging at DEBUG or TRACE level in production — the volume can degrade performance and overwhelm log storage systems.

Monitoring Frameworks: Prometheus & Grafana

Observability is the ability to understand the internal state of a system from its external outputs — metrics, logs, and traces. In production environments, especially microservices architectures, monitoring is not optional. Teams need real-time visibility into application health, performance characteristics, and resource utilization to detect anomalies, prevent outages, and optimize performance. Prometheus and Grafana are the industry-standard open-source monitoring stack for Java applications.

Prometheus is a time-series database and monitoring system that periodically scrapes metrics endpoints exposed by application instances. Spring Boot Actuator exposes a `/actuator/prometheus` endpoint that Prometheus can scrape, providing hundreds of JVM and application metrics out of the box. Grafana connects to Prometheus as a data source and provides a powerful, flexible dashboard interface for visualizing those metrics in real time.

Response Time

Track P50, P95, and P99 response time percentiles to identify performance degradation before users are impacted.

Error Rates

Monitor HTTP 4xx and 5xx error rates to detect application failures and broken integrations immediately.

Throughput

Measure requests per second to understand traffic patterns and capacity requirements for scaling decisions.

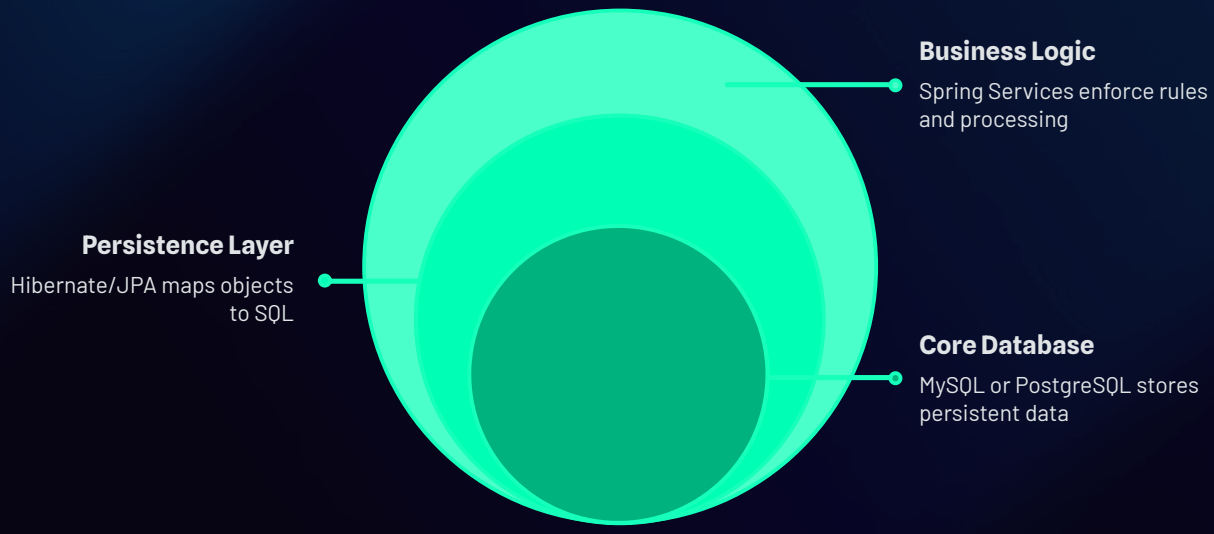
Resource Utilization

Track JVM heap usage, CPU consumption, thread pools, and database connection pools to prevent resource exhaustion.

Configure Prometheus alerting rules to automatically notify your team via Slack, PagerDuty, or email when metrics exceed defined thresholds — enabling proactive incident response.

Full Stack Architecture: Enterprise Application Flow

A production-grade Java full stack application is composed of multiple interconnected layers, each with a clearly defined responsibility. Understanding how data flows through the entire stack — from the moment a user clicks a button in the browser to the moment a response is rendered on screen — is essential for diagnosing issues, designing new features, and making informed architectural decisions. The layered architecture also provides natural boundaries for security controls, performance optimization, and team organization.



Each layer communicates exclusively with its adjacent layers, enforcing the separation of concerns principle at the architectural level. The UI layer communicates with the REST API layer over HTTPS using JSON. The controller layer delegates to service classes containing business logic. Services interact with repository interfaces (Spring Data JPA) that Hibernate translates into optimized SQL queries against the database. This clean separation means individual layers can be tested, replaced, or scaled independently without affecting the rest of the system.

Presentation Tier

Angular or React applications handle rendering, user input, and state management entirely in the browser, communicating with back-end via REST APIs.

Application Tier

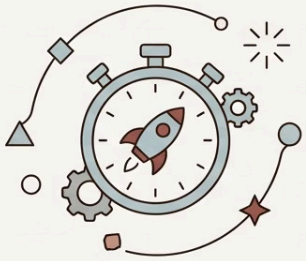
Spring Boot services handle request routing, business rule execution, transaction management, and security enforcement on the server side.

Data Tier

Relational databases (MySQL, PostgreSQL) or NoSQL stores (MongoDB) persist application state, with Hibernate managing the object-relational translation layer.

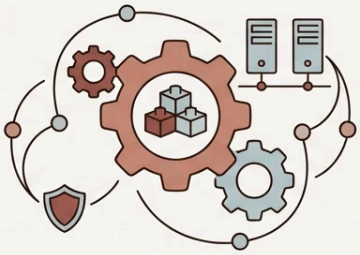
Framework Selection Guidelines

Choosing the right combination of frameworks is one of the most consequential architectural decisions a development team makes. The wrong choice can result in unnecessary complexity, steep learning curves, and frameworks that fight against rather than support your application's requirements. The right choice accelerates development, reduces operational burden, and scales gracefully with your application's growth. Framework selection should be driven by project scale, team expertise, performance requirements, and long-term maintainability goals.



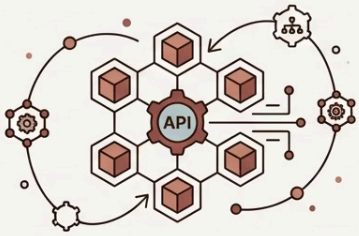
FOR SMALL PROJECTS

- ☑ Backend: **Spring Boot**
- ☑ Frontend: **Angular** or **React**
- ☑ Database: **MySQL**
- ☑ Builds: **Maven**
- ☑ Notes: **Simple & Fast**



FOR ENTERPRISE APPLICATIONS

- ☑ Framework: **Spring Boot + Cloud**
- ☑ Frontend: **Angular**
- ☑ ORM: **Hibernate JPA**
- ☑ Database: **PostgreSQL**
- ☑ Orchestration: **Kubernetes**
- ☑ CI/CD: **Jenkins**
- ☑ Notes: **Full Featured, Scalable**



FOR MICROSERVICES PLATFORMS

- ☑ Framework: **Spring Cloud**
- ☑ Services: **Multiple Services**
- ☑ Containers: **Docker**
- ☑ Orchestration: **Kubernetes**
- ☑ Gateway: **API Gateway**
- ☑ Monitoring: **Prometheus**
- ☑ Notes: **Distributed, Resilient**

Key Selection Criteria

- **Team Expertise** — Choose frameworks your team already knows or can learn quickly
- **Project Scale** — Small APIs don't need Kubernetes; large platforms do
- **Performance Requirements** — Consider reactive frameworks (Spring WebFlux) for high-concurrency scenarios
- **Community & Support** — Active communities mean better documentation, tutorials, and faster bug fixes

Common Anti-Patterns to Avoid

- Over-engineering small projects with microservices prematurely
- Using multiple incompatible ORM frameworks in a single project
- Skipping security frameworks in favor of custom implementations
- Neglecting build automation in favor of manual deployment processes
- Adopting bleeding-edge frameworks without considering long-term support

Quick Revision: Java Full Stack Framework Summary

This reference card consolidates all the essential frameworks covered in this guide for rapid revision. Whether preparing for an interview, an examination, or a technical discussion, use this summary to quickly recall the key tools across every layer of the Java full stack. Each framework was chosen for its industry relevance, community adoption, and integration within the broader Spring ecosystem.

🖥️ Front-End

- **Angular** — Enterprise SPAs, TypeScript-based
- **React** — Flexible UI library, Virtual DOM
- **Vue.js** — Progressive, lightweight
- **Bootstrap** — Responsive CSS components
- **Tailwind CSS** — Utility-first styling

⚙️ Back-End

- **Spring Framework** — Core IoC/DI container
- **Spring Boot** — Auto-configuration, embedded servers
- **Spring MVC** — Web MVC pattern
- **Spring Security** — Auth & authorization
- **Spring Cloud** — Microservices infrastructure

🗄️ Data & APIs

- **Hibernate** — ORM framework
- **JPA** — Persistence specification
- **Spring REST** — RESTful API development
- **JWT / OAuth2** — API security tokens

🚀 DevOps & Testing

- **JUnit 5** — Unit testing framework
- **Mockito** — Dependency mocking
- **Maven / Gradle** — Build automation
- **Docker** — Containerization
- **Kubernetes** — Container orchestration
- **Jenkins** — CI/CD pipelines

Master the Spring ecosystem first — Spring Boot, Spring MVC, Spring Security, and Spring Data JPA cover the majority of enterprise Java development requirements and are the most commonly tested topics in Java full stack interviews and certifications.

GSDC
Global Skill Development Council

CERTIFIED JAVA FULL STACK DEVELOPER (CFSD)

ABOUT GSDC CERTIFICATION

EBOOK

Extensive and exclusive Ebook created by world's experts to help you with understanding core concepts.

LEARNING MATERIALS

Get access to learning materials such as videos, ebooks, templates, and practice exams, which will help you clear the certification exam.

CREATED BY EXPERTS

GSDC certifications are created and authored by world's leading experts in the field.

LEARNING OBJECTIVE

- Gain insights into autonomous decision-making processes
- Apply knowledge using ready-to-implement templates
- Demonstrate ability to work with Agentic AI models
- Validate your skills wit

Enroll now with the code **LEARN20** To avail **20%** discount

Enroll Now

www.gsdcouncil.org