**Agile**

It is a flexible, iterative, approach to software development. Emphasizes collaboration, customer feedback, and small, rapid release.

**Scrum** : Most common agile framework used in IT companies.

**Sprint** : A short, time-boxed period (1-4 weeks ) where a team works to complete a set of tasks.

**User Story** : A short, simple description of a feature told from the perspective of the user.

**Backlog** : A prioritized list of work (user stories, bugs, enhancements) to be done.

**Agile Roles :**

**Product Owner (PO)** : Manages the Product Backlog, Represents the voice of the customer, prioritizes what to build next.

**Scrum Master** : Ensures the team follows Agile/Scrum principles, removes roadblocks, facilitator, not a manager.

**Development Team** : Cross-functional members who design, develop, test, and deliver the product.

**Key Meetings :**

1. **Product backlog Preparation:**

Product Owner collects requirements from stakeholders and breaks them into **User Stories** and maintains the **Product Backlog**.

**Activity** : **Backlog Refinement Meeting** :

Held by **Product Owner + Dev Team** regularly once a week during the sprint to review to discuss upcoming stories, clarify requirements, and estimates story points and break big items (epic) into smaller stories.

**Outcome** : well-prepared backlog ready for Sprint Planning.

1. **Sprint Planning Meeting :**

held by **Scrum Master, Product Owner, Development Team**. at the **beginning of every sprint for 2- 4 hours** depending on the sprint length to plan the upcoming sprint

**Activities :**

Sprint Goal – PO proposes a goal the sprint should accomplish.

Story Selection – Team selects stories from the top of the backlog.

Estimation Recheck – Confirm/revise story points estimate if needed.

Task Breakdown – Break each story into subtasks (e.g. design, code, test).

Capacity Check – Team checks availability (e.g. leaves, holidays).

**Outcome :** Sprint Backlog, Sprint Goal and Tasks with clear owners.

1. **Daily Scrum (Stand-up Meeting)**

Held by **entire Scrum Team** where usually **developers and testers only speak**. It is done every day for **maximum 15 minutes** to **synchronize the team and identify backlog early.**

Discussions on topic like What did I do yesterday? What will I do today? Is there anything blocking me? The Scrum Master notes idempotents and helps resolve them quickly.

1. **Sprint Execution :**

Here developers code, QA testers tests in parallel and bugs are fixed as they are found. They use Kanban/Jira boards to move tasks – To Do → In Progress → In QA → Done.

Continuous Integration (CI) may be used.

The team adapts if priorities shift or blockers occur, but **no new work** is added mid-sprint **unless its critical.**

1. **Sprint Review Meeting :**

This is **end of the sprint** held for 1 – 2 hours by **Scrum Team + Stakeholders** (clients, managers)

Activities :

It is done to demo the completed feature where **PO confirms** if **Definition of Done (DoD)** is met. Stakeholders ask questions or give feedback and **new ideas may be added** to the backlog. The feedback is collected and incorporated in future planning.

1. **Sprint Retrospective Meeting**

It is done after the Sprint review / end of the Sprint for – 1.5 hrs by **Scrum Team only** (PO, Scrum Master, Devs).

**Discussion like – What should we start/stop/continue doing?**

🟢 What Went Well, 🔴 What Didn’t, 🟡 Action Items

2-3 action items to improve team collaboration and sprint efficiency, Team commits to trying those in the next sprint.

**Activities :**

**Definition of Done (DoD)**  Agreed criteria for when a task/story is “done”. E.g. coded, tested, documented, code reviewed, merged etc

**Definition of Ready (DoR)** Criteria for the story to be ready to pick, E.g. well-defined, estimated, acceptance criteria clear)

**Velocity Tracking** Track how many story points the team completes per sprint to estimate future work capacity

**Burndown Chart** Graph that shows how much work remains in the sprint over time

Idempotent Log Scrum Master tracks and remove team backers

**Tools Used** : Jira/ Azure Boards for backlog management, sprint planning, tracking

Git / GitHub/ GitLab - Source control

Environments use in Software development and development pipelines:

**DEV (Development)**

Where developers write and test their code. Used by developers.

It’s the earliest stage, with frequent changes and deployments. Developers write features, fix bugs, and push code here often. Fast feedback, no strict rules.

Bugs are expected here – It’s basically your playground.

**SIT (System Integration Testing)**

Code from DEV is deployed here for tester teams to run full workflow tests, execute test scripts, and identify the issues.

This environment tests how various system components (APIs, databases, services) work together.

It’s more stable than DEV and used occasionally by developers for debugging, but less refined than UAT.

**UAT (User Acceptance Testing)**

Done by end users or business stakeholders.

Goal: Confirm the system works as they expect before going live.

End users or product owners test real-world scenarios. Feedback here decides if the product is ready. This environment should closely mirror production.

**PROD (Production)**

The live environment where real users interact with the application.

Once everything is approved, code is deployed here. Monitored carefully. Issues here are critical.

Highest level of stability, monitoring, and performance is expected.

Any issues here can directly impact users.

**Quick Analogy: Building a House**

* **DEV** = You’re designing and trying different things on a model.
* **SIT** = You test plumbing, electricity, doors — does it all work together?
* **UAT** = The homeowner walks through and says, “Yep, this is what I wanted.”
* **PROD** = The family moves in and lives there.

**Loggers :**

Logging is an essential part of application development that allows developers to monitor and troubleshoot their applications. Alternative of System.out.println() – more powerful and configurable.

Why? Helps trace issue in production, allows log level control (e.g. WARN, ERROR), supports writing logs to files, console, database, or remote monitoring systems.

Spring Boot support various logging frameworks, such as Logback, Log4j2, and Java Util Logging (JUL)

* **LogBack (preferred – mostly used):** A popular logging framework that serves as the default in many Spring Boot applications. It offers a flexible configuration and good performance
* **Log4j2 (for advance features):** Another widely used logging framework with features such as async logging and supports for various output formats.
* **Java Util Logging (Jul) (Present inside java not for advance features):** The default logging framework included in the Java Standard Edition. Write its less feature-rich than some third-party frameworks, it is straightforward and is a part of the Java platform

Spring Boot comes with a default logging configuration that uses LogBack as default logging implementation. It provides a good balance simplicity and flexibility. The default configuration is embedded within the Spring Boot libraries, and it may not be visible in your project’s source code.

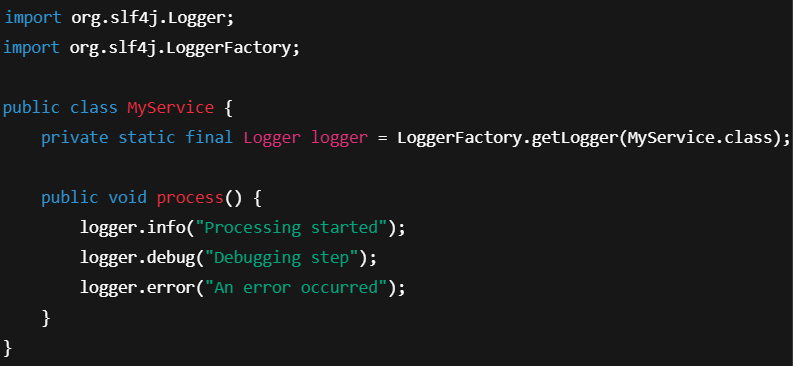
If you want to customize the logging configuration, you can create your own logback spring.xml or logback.xml file in the src/main/resource directory. When Spring Boot detects this file in your project, it will use it instead of the default configuration.

**TRACE DEBUG INFO WARN ERROR**

We can set the desired logging level for specific packages or classes, allowing them to control the amount of information logged at runtime.

By default, logging is enabled for INFO, WARN, ERROR

Spring Boot provides annotations like @Slf4j & @Log4j2 that you can use to automatically inject logger instance into your classes.

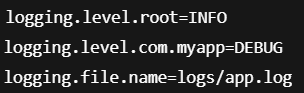


**Every class has its private instance and so for the logger that’s why we use class name**

**We want only single instance of logger that’s why make it private static final**

**Spring Boot Logging Configuration**

In application.properties :



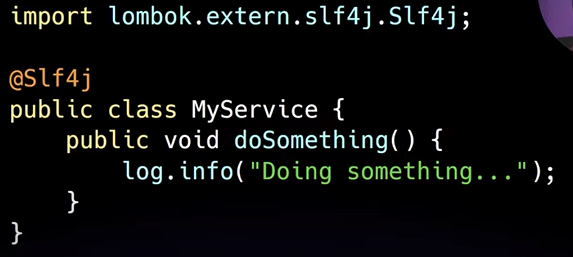
**Best Practices**

* Use parameterised logging :

 // Use placeholder to insert value

* Avoid logging sensitive data (e.g., passwords).
* Do not use System.out.println() in production code.
* Use DEBUG for development, INFO/WARN/ERROR for production.
* Always keep ERROR in catch block.

By using Slf4j



**Appenders :**

**Appenders** are responsible for writing log messages to various destinations

**Appender = Destination for log messages**  
Examples: console, file, database, email, etc.

**Type of appenders :**

| **Appender Type** | **Description** |
| --- | --- |
| ConsoleAppender | Sends logs to the console (terminal or IDE). |
| FileAppender | Writes logs to a specified file. |
| RollingFileAppender | Writes to a file and rolls it over based on size/time. |
| DailyRollingFileAppender | Rolls logs daily (Log4j 1.x). |

**Questions :**

**How do you resolve production bugs?**

There are multiple ways to approach and resolve production bugs, depending on the situation.

When bug Is raised in the production then I have to analyze the logs if I am able to understand the root cause of the bug then I need to fix that code accordingly and I have to push and deploy that code.

If from the logs I am not able to get the root cause then I first coordinate with the tester to see if the issue can be reproduced in a lower environment such as SIT or DEV.  
If the bug is reproducible there, I try to replicate it in my local environment. Once I'm able to reproduce it locally, I analyze the code, identify the root cause, and apply the necessary fix.

However, if the bug **cannot be reproduced in lower environments**, I move to the next step.  
In such cases, I increase the logger level to INFO or DEBUG in the production environment (if allowed) to capture more detailed logs. These logs help in understanding what might be going wrong in the production setup and guide me toward identifying the root cause. Based on this insight, I implement the fix and validate it thoroughly before pushing it to production.

**What logging framework does Spring Boot use by default?**

**What's the difference between SLF4J and Logback?**

Think of SLF4J (façade – abstraction) as an interface and Logback as the implementation.

**How you see the logs?**

**There are multiple ways**

So, there is a terminal option in Microsoft Azure and there is a download option from which we can download the logs.

Another way is there a WinSCP to access the server so through WinSCP also we download logs sometimes

Add the log configuration such that logs will be directly send to Elastic Search. We are planning to integrate with the Elastic Search.

**OR**

In current project, we are using WinSCP to manually download logs from the server. Now we are planning to use Elastic Search to monitor the logs. It provides more flexibility and allows us to retrieve logs based on specific dates.

**What is the purpose of using a logging facade like SLF4J?**

SLF4J provides a standard, unified API for logging, decoupling your code from the actual logging implementation.

* You can **easily switch** between Logback, Log4j2, etc. without changing logging code.
* Promotes **cleaner**, more **maintainable** code.
* Supports **parameterized logging** to avoid unnecessary string concatenation.

**How do you change the logging level in Spring Boot?**

**What are some best practices for logging in a production application?**

**Where the code is stored?**

In Agile (or any software development process), the \*\*code is stored in a **Version Control System (VCS)** — most commonly **Git**.

**Sprint last for how many days?**

For 1 – 4 weeks

**how you estimate your stories?**

Estimating **user stories in Agile/Scrum** means designing how much effort a user story will take, using relative scale rather than exact time.

* You estimate the **relative effort** (not hours) using a scale—often **Fibonacci** (1, 2, 3, 5, 8, 13...).
* “1” means small and easy and “13” means large and complex.
* Team Discussion (Planning Poker) is done where everyone gives their estimate, if people disagree, they explain why, then re-vote until they agree.
* If you know Story A took 5 points, compare new stories to that one. It’s all relative – not exact science.
* After few sprints, your team knows how many points they can finish in one sprint, this helps plan how much can fit in the next sprint.

**What is Spring Profile?**

Spring Profile is a feature that lets you manage environment-specific configurations and control which beans are loaded based on the active environment

.

**How you manage Spring profile ?**

In Spring Boot, profiles are used to manage configuration and bean registration separately for different environments like development, testing, staging, and production. We typically create environment-specific property files such as **application-dev.properties**, **application-test.properties**, and **application-prod.properties**, while keeping **common properties in** **application.properties**. The active profile can be set in several ways — inside application.properties, through a command-line argument (--**spring.profiles.active=dev**), or as an environment variable, depending on the deployment strategy. Spring automatically merges the base application.properties with the active profile's properties, giving priority to profile-specific values when there’s a conflict. **To control bean registration**, we can use the **@Profile** annotation on **configuration classes or beans**, ensuring that only the relevant components are loaded for the active environment. We can also **activate multiple profiles simultaneously**, which allows fine-grained control over configuration layering. If no profile is set, Spring uses the **default** profile. Additionally, it's important to note that profiles are also supported in **application.yml**, offering better organization for **complex configurations**. Managing profiles properly ensures clean separation of concerns, minimizes errors during environment transitions, and simplifies DevOps and deployment processes.

* Spring supports a **default profile** named default if none is explicitly activated.
* You can create **profile-specific beans** inside a @Configuration class using @Profile.
* You can **nest** properties inside application.yml using --- to define multiple profiles in a single file.
* In Spring 3.1+, you can use **@Profile("!prod")** to indicate beans that should load when a profile is **not** active.
* Profile information can also be injected using **@Value("${spring.profiles.active}")** if needed at runtime.

1. **application.properties** (common for all)

spring.profiles.active=dev

2. **application-dev.properties** (for dev environment)

server.port=8081

app.message=Hello from Dev Environment!

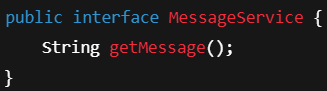
**application-prod.properties** (for prod environment)

server.port=8080

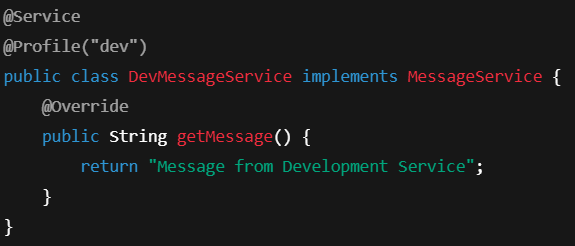
app.message=Hello from Production Environment!

3. **Java Code with @Profile**

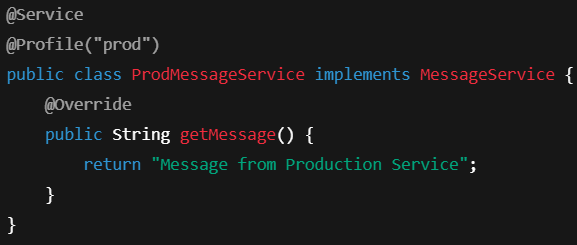
Suppose you have two service classes and you want to load different ones based on the profile.



Development Service Class



Production Service Class



4. **Controller to test**



If you run the app with **spring.profiles.active=dev**, Spring will inject DevMessageService.

If you switch it to **prod**, Spring will inject ProdMessageService.

You can call localhost:8081/message (for dev) or localhost:8080/message (for prod) to test it.

**How deployment is done in your project?**

**Deployment Process Overview:**

In our project, deployment is done through a CI/CD pipeline using tools like GitHub Actions, Jenkins, or GitLab CI. Once the code is pushed to the main branch or a release branch, the pipeline is triggered automatically.

**Steps Involved:**

1. **Build:**  
   The code is built using Maven. This includes compiling the source code, running unit tests (e.g., JUnit), and packaging the application as a JAR or WAR.
2. **Static Code Analysis:**  
   Tools like **SonarQube** and **Checkstyle** are used to analyze code quality and enforce coding standards before proceeding.
3. **Containerization:**  
   The application is packaged into a Docker container using a Dockerfile. This makes it portable and consistent across environments.
4. **Pushing to Registry:**  
   The Docker image is pushed to a container registry like **Docker Hub, GitHub Container Registry,** or **Azure Container Registry.**
5. **Deployment:**  
   The image is then deployed to a target environment:
   * **Local or Test:** Using **Docker Compose** or **Kubernetes** (Minikube).
   * **Staging/Production:** Using **Kubernetes** (OpenShift on Azure) or **cloud services** like Microsoft Azure (e.g., Azure Kubernetes Service (AKS), Azure App Services, etc.).
6. **Monitoring & Logging:**  
   Tools like **Prometheus, Grafana, ELK stack**, or **Spring Boot Actuator** are used to monitor the deployed application.

We are using Jenkins pipeline in OpenShift/Microsoft Azure environment

This approach ensures **automated, reliable, and repeatable deployments**, reduces manual errors, and supports rapid delivery of updates.

**How you maintain your code quality?**

code review, then Junit, SonarCube

**Code Review** is a manual inspection of code by peers during pull requests to ensure readability, maintainability, and adherence to coding standards. It helps catch logical errors early, improves design, and promotes team collaboration and knowledge sharing.

**JUnit Testing** is used to validate individual components of a Java application. Writing unit tests with JUnit (and Mockito for mocking) ensures that methods behave as expected, facilitates safe refactoring, and improves code reliability by catching bugs early.

**SonarQube** is a static code analysis tool that identifies bugs, vulnerabilities, code smells, and checks test coverage In current product we are trying to cover atleast 80% of JUnit code coverage. Integrated with CI/CD pipelines, it enforces quality gates to prevent low-quality code from being merged, supporting continuous improvement and clean code practices.

**1. Clean Code Principle**

Clean Code refers to writing code that is:

* **Readable**: Easy to understand by humans.
* **Maintainable**: Easy to change, fix, or extend.
* **Consistent**: Follows a uniform coding style and naming convention.
* **Simple**: Does not add unnecessary complexity.

Key practices:

* Meaningful names for variables,+ methods, classes.
* Small, focused functions.
* Avoid duplication.
* Use comments **only** where necessary.
* Follow a clear structure and formatting.

**2. DRY Principle (Don't Repeat Yourself)**

**Goal**: Eliminate duplication.

If the same logic or code is repeated:

* Extract it into a **function**, **class**, or **module**.
* Reuse it wherever needed.

**Why?**

* Easier to maintain and update.
* Reduces bugs from inconsistent changes.

**3. YAGNI Principle (You Aren’t Gonna Need It)**

**Goal**: Avoid writing code for features unless they are actually needed.

Don’t build functionality based on assumptions about future requirements.

**Why?**

* Saves time and effort.
* Reduces complexity.
* Prevents overengineering.

**4. KISS (Kee it Simple, Stupid)**

Used to avoid overcomplicating code or design.

Write simple, clear, and easy-to-understand solutions.

**5. L in SOLID – Liskov Substitution Principle**

Part of the **SOLID** principles (object-oriented design).

**Definition**:  
*Subtypes must be substitutable for their base types without altering the correctness of the program.*

**In other words**: If S is a subclass of T, then objects of type T should be replaceable with objects of type S **without breaking** the functionality.

**Violations often include**:

* Overriding methods in a way that breaks expectations.
* Throwing exceptions or altering behavior not expected by base class consumers.

**mock inject, mock, and annotation used?**

**@Mock**  
Used to create mock objects instead of calling real implementations.  
Commonly used for collaborators like repositories, services, or external APIs.

**@InjectMocks**  
Used to inject mock objects (annotated with @Mock) into the class under test.  
The class under test must have dependencies (e.g., via constructor or field) for this to work.

**@RunWith(MockitoJUnitRunner.class)** (for **JUnit 4**)  
**@ExtendWith(MockitoExtension.class)** (for **JUnit 5**)  
These annotations enable the processing of @Mock and @InjectMocks.

**Installing Eclemma:**

Help -> install new softwares -> work with : <https://download.eclipse.org/eclemma/releases/> -> select Eclemma -> select all -> trust all -> next -> next -> finish

**What will you do if the task is assigned to you?**

If suddenly any task is assigned to me and it is on the priority then I have to discuss with scrum master to generate the story for me and then I will work on that story.

**If any bug is assigned to me?**

Then I have to analyse the code if there is any risk then I have to get the clarity from the resprctive person if there is no risk then I have to work on that and then after development I have to raise the pr and reviewer will review that code then I have to merge that code in the main branch and deploy that on the server.