Capgemini

Java Delivery Playbook Document

Table of Contents

Purpose of this document

Provides:

Describe the approach to overall delivery of a typical Java project for a subset of the organization(wave) that is embarking upon larger transformation program.

Provide guidelines/guardrails on activities to be performed as part of overall Delivery.

- Possible approach to various phase of implementation
- Guidelines and roadmaps to be followed for a selected SDLC cycle.
- Phase planning, tool sets to be used.
- Methodology and checklists

Not provide:

- Does not explore the benefits and rationale for adopting a particular way of delivery cycle.
- Does not provide a holistic compendium/how to guide on the Delivery process.

This playbook is a living document that will evolve as the team continues to embark of new projects and learn through its implementation.

Software Delivery Journey

This documentation provides helpful guides and documentation for engineers to work on the tasks involved in software delivery. The information is split into following categories:

- Technical guides
- Reference documentation for tooling
- Technical How Tos
- Playbooks to explain end-to-end.

Planning:

- The development journey starts with requirements from business, which is then translated to workable technical work by Product Owners and Architects
- The tooling used for planning, tracking and Managing is JIRA, Rally ..etc.

Getting Started:

1. Download Spring or Quarkus from the official websites
    a. https://start.spring.io/
    b. https://code.quarkus.io/
2. Establish the coding conventions in the team
3. Align the architecture of the new service with an architecture pattern

Selection of Project Life Cycle:

Linear Development Lifecycle (LDM)

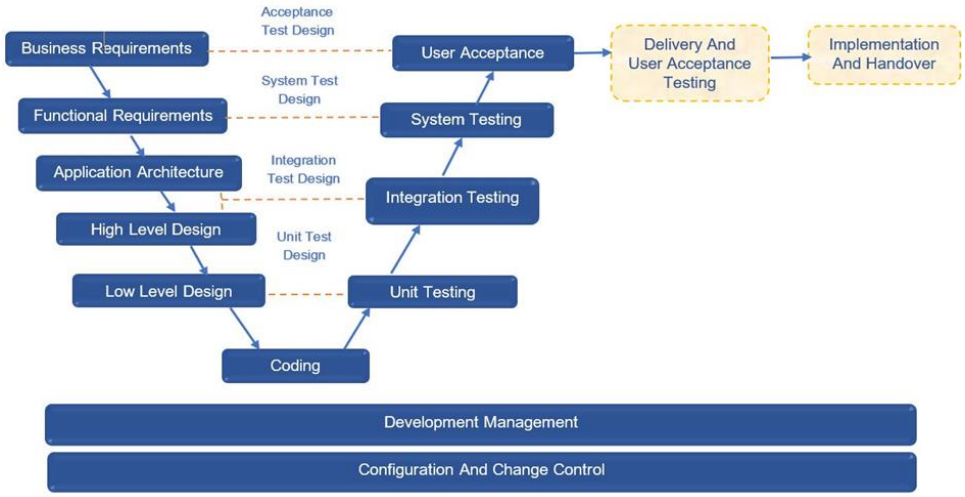
Overview:

The Linear Development Method is the Capgemini recommended lifecycle method for development engagements executed in waterfall/v-model. LDM is designed to support application development where there are clearly stated business requirements.
Linear Development Method is structured as a progressive, deliverables-driven life cycle that is divided into successive phases. In this method, both development team and test teams work in parallel and final product is produced at the end.

Lifecycle selection criteria:

- Requirements are well defined, clearly documented and fixed.
- Product definition is stable.
- Technology is not dynamic and is well understood by the project team.
- There are no ambiguous or undefined requirements.
- The project is short.

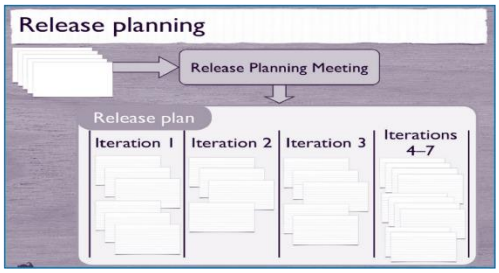
LDM Lifecycle phases:



Refer - https://capgemini.sharepoint.com/sites/QAIMS/SEPG/QMS_Wrapper/SitePages/LDM%20Wrapper.aspx

Agile Delivery Lifecycle – Cap Gemini Agile Framework (CAF)

Agile planning covers three different levels—the release, the iteration, and the current day—each plan can be made with a different level of precision thus enabling a wholistic planning view



Refer - https://capgemini.sharepoint.com/sites/SSO10/SitePages/Agile-CAF-Lifecycle.aspx

Development:

Along with industry standard tooling, there are some recommended platforms that support the development and CICD journey for engineers.

Continuous Integration

Software development

- Code development:

- Code development would be done via IDE IntelliJ, Eclipse.
- It can be test driven development to resolve some coding bugs.

Build process – Used for building Java projects.

- Gradle :
  - https://spring.io.guides/gs/gradle/
  - IntelliJ set up for Gradle https://www.jetbrains/help/idea/gradle.html
  - Sample gradle configuration
- Maven:
  - https://sping.io.guide/gs/maven/
  - IntelliJ set up for Maven https://www.jetbrains.com/help/idea/maven.html
  - Sample pom configuration

https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

  - Git and GitHub- These guides cover new repo creation, branch permissions and creation and other features.
  - Jenkins
    - Jenkins Pipeline – Jenkins pipeline is a suite of plugins that supports implementing and integrating continuous delivery pipelines into Jenkins.

Recommended Build tools

Maven (POM):

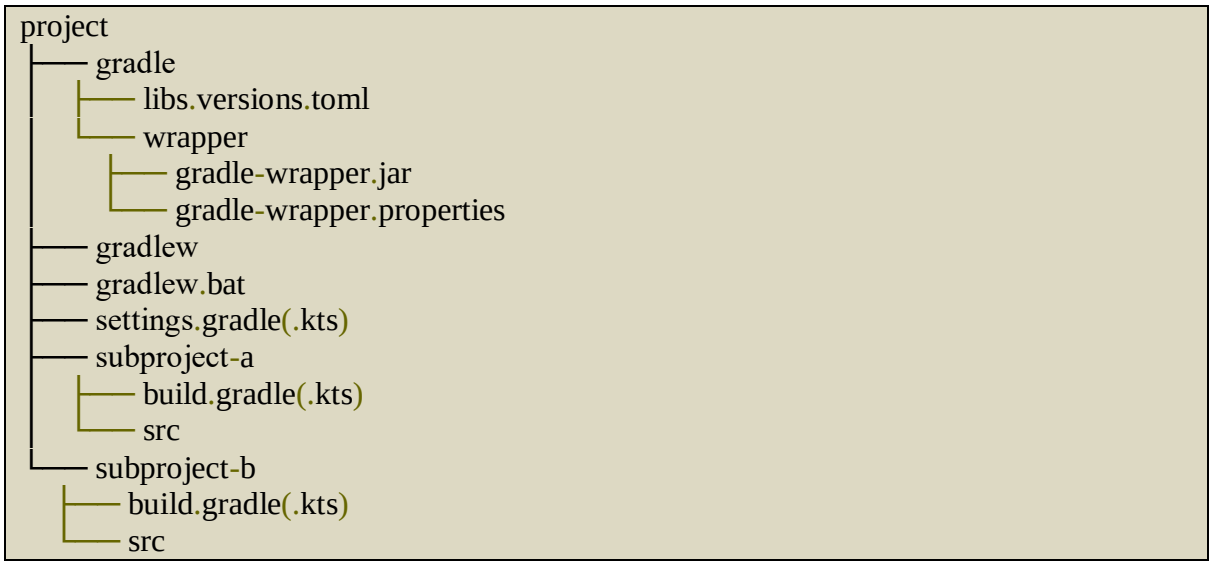
- A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project
- Minimal POM: The minimum requirement for a POM are the following:
  - project root
  - modelVersion - should be set to 4.0.0
  - groupId - the id of the project's group.
  - artifactId - the id of the artifact (project)
  - version - the version of the artifact under the specified group
- Sample pom

```
1. <project>
2. <modelVersion>4.0.0</modelVersion>
3.
4. <groupId>com.mycompany.app</groupId>
5. <artifactId>my-app</artifactId>
6. <version>1</version>
7. </project>
```

Refer - https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

Gradle:

- Like we have pom.xml with Maven, we have build.gradle with Gradle.
- A Gradle project will look something like:



Refer - https://docs.gradle.org/current/userguide/getting_started_eng.html

Additional Development Tooling:

Include tools that contribute in static analysis and code security scanning aspects:

- Nexus – Distribute packaged and containerized apps like Docker, Helm, Yum and APT
- SonarQube – Continuous inspection of code code quality to detect bugs and code smells.
- NexusIQ – Scan software libraries(3rd party, open source and custom) to create a detailed inventory of the components that comprise your applications. It constantly monitors inventoried components for new risks

Coding Conventions:

In an effort to continuously push developers toward "a standard way to develop", we recommend to refer –
https://google.github.io/styleguide/javaguide.html
https://www.oracle.com/technetwork/java/codeconventions-150003.pdf
https://devonfw.com/docs/java/current/cross_cutting/coding_conventions.html

In this the below topics are covered in the coding standards

- File Names
- File Organization
- Indentation
- Comments
- Declarations
- Statements
- White Space
- Naming Conventions for class, method, variables used by the Java program

Naming Conventions:

We recommend using the following naming conventions for services, object, functions, programs, tables, files etc as they are means by which we identify things
Refer- https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html
The naming standards are focussed on following areas:

- XML namespaces
- Java packages
- EJB Object Names
- JNDI names
- CORBA Object Names
- Constants

Java Application Logging Framework

Logging equips the developer with detailed context for application failures. To align with the global /
Refer: https://logging.apache.org/log4j/2.x/manual/index.html

Junit Unit Testing Framework

Unit tests should be done to ensure the functionality and behaviour of individual software component prior to any system level testing. Junit, an open source, light-weighted framework for Java application unit testing.
Refer – https://junit.org

- It allows developer to write test code faster while increasing quality
- It is elegantly simple and inexpensive.
- It checks their own results and provides immediate feedback
- It can be composed into a hierarchy of tests suites

Exception Handling

To preserve the readability, maintainability and extensibility of our code, it's a good idea to create our own subtree of application-specific exceptions that extend the base exception class. Hence it is always recommended to have your own Application Error hierarchy
Refer:

- https://www.javaguides.net/2018/06/guide-to-java-exception-handling-best-practices.html

Back-End Java

Best Practices:

Validate Inputs and handle errors:

You can validate backend inputs at two levels:

1. **API gateway** — the validation of inputs at the API gateway level via policies; mainly generic validation, i.e. schema, format.
2. **Microservice** — the microservice level validations involve the checking of the existence of entities, etc. There are libraries, e.g. Joi validator, that you can leverage on for ease of input validation depending on your development stack.

Handle error using:

You can consider using two methods to handle errors to mitigate the downstream impact

1. **Circuit breaker —** prevents repeated invocation of services that are likely to fail. There are available libraries and examples for NodeJS, Springcloud, etc. which you can implement the pattern quickly.
2. **Handle exceptions with error codes** — returns a response without crashing the service with a business or HTTP error code to facilitate troubleshooting.

Unit Testing:

Source Code:

It's a good idea to keep the test classes separate from the main source code. So, they are developed, executed and maintained separately from the production code.
Also, it avoids any possibility of running test code in the production environment.
We can follow the steps of the build tools such as Maven and Gradle that look for src/main/test directory for test implementations.

Package Naming Conventions:

We should create a similar package structure in the src/main/test directory for test classes, this way improving the readability and maintainability of the test code**.**

Test Case Naming Conventions:

The test names should be insightful, and users should understand the behaviour and expectation of the test by just glancing at the name itself.
It's often helpful to name the test cases in given_when_then to elaborate on the purpose of a unit test
We should also describe code blocks in the *Given*, *When* and *Then* format. In addition, it helps to differentiate the test into three parts: input, action and output

Expected Vs Actual:

A test case should have an assertion between expected and actual values.
To corroborate the idea of the expected vs actual values, we can look at the definition of the *assertEquals* method of JUnit's *Assert* class:
public static void assertEquals(Object expected, Object actual)
It's suggested to prefix the variable names with the actual and expected keyword to improve the readability of the test code.

Appropriate Annotations:

Always use proper assertions to verify the expected vs. actual results. We should use various methods available in the *Assert* class of JUnit or similar frameworks such as AssertJ.
For eg: Assert.assertEquals, assertNotEquals, assertNotNull, assertTrue

Specific Unit Tests:

Instead of adding multiple assertions to the same unit test, we should create separate test cases.
Always write a unit test to test a single specific scenario

Test Production Scenarios:

Write tests considering real scenarios in mind, it helps to make unit tests more relatable

Mock External Services:

We should mock the external services and merely test the logic and execution of our code for varying scenarios. We can use various frameworks such as Mockito, EasyMock and JMockit for mocking external services

Avoid Code Redundancy:

Create more and more helper functions to generate the commonly used objects and mock the data or external services for similar unit tests. This enhances the readability and maintainability of the test code

Annotations:

We should leverage annotations to prepare the system for tests by creating data, arranging objects and dropping all of it after every test to keep test cases isolated from each other. Often, testing frameworks provide annotations for various purposes, for example, performing setup, executing code before and tearing down after running a test. Various annotations such as JUnit's *@Before*, *@BeforeClass and @After* and from other test frameworks such as TestNG are at our disposal.

80% Test Coverage:

As a rule of thumb, we should try to cover 80% of the code by unit tests. Additionally, we can use tools such as JaCoCo and Cobertura along with Maven or Gradle to generate code coverage reports

Coverage:

1. **Jacoco**: JaCoCo stands for Java Code Coverage. It is a free code coverage library for Java, which has been created by the EclEmma team. It creates code coverage reports and integrates well with IDEs like IntelliJ IDEA, Eclipse IDE, etc. JaCoCo also integrates with CI/CD tools like Jenkins, Circle CI, etc., and project management tools like SonarQube, etc.
   https://www.geeksforgeeks.org/how-to-generate-code-coverage-report-with-jacoco-in-java-application/

2. **Sonar:**

   SonarQube is the leading tool for continuously inspecting the Code Quality and Security of your codebases, and guiding development teams during Code Reviews
   https://docs.sonarsource.com/sonarqube/latest/try-out-sonarqube/

Contract driven testing:

Contract testing is about making sure your consumer team and provider team have a shared understanding of what the requests and responses will be in each possible scenarios

- Which consumer-side tests specifications should I write?
  - For a given service, for a given end point, write a contract test for all methods/responses status combinations, unless your service explicitly returns 401 and 403, you don't need to test this
  - Headers when they are specific to the service
  - In return response, validate only that the information your consumer cares about was delivered in the format you are expecting

Monitoring:

This term is also referred as APM i.e. Application performance monitoring. Using application performance monitoring (APM) solutions, businesses can monitor whether their IT environment meets performance standards, identify bugs and potential issues, and provide flawless user experiences via close monitoring of IT resources. In short, end-to-end application performance monitoring works by:

- Observing whether your apps are behaving normally.
- If not, alerting to and collecting data on the source of the problem (be it the app, app dependencies, or supporting infrastructure)
- Analysing the data in the context of the impact on the business
- Adapting your application environment to fix similar problems before they impact the business.

Few of the most critical application performance monitoring metrics:

- CPU usage: At the server level, APM looks at CPU usage, memory demands, and disk read/write speeds to make sure usage doesn't affect app performance.

- Error rates: At the software level, APM tracks how often app performance degrades or fails. For example, when web requests end in an error or during memory-intensive processes like searching a database.
- Response times: Average Response Time is the metric that shows whether speed is affecting app performance.
- Number of instances: For elastic, cloud-based applications, you need to know how many server or app instances you have running at any one time. APM solutions that support autoscaling can then cost-effectively scale your app to meet user demand.
- Request rates: This metric measures how much traffic your application receives — any spikes, inactivity, or numbers of concurrent users.
- Application availability/uptime: This metric, which monitors whether your app is online and available, is the one most enterprises use to check compliance with SLAs.
- Garbage collection (GC): If your app is written Java or another programming language that uses GC, you'll be all too familiar with the problems that arise from its heavy use of memory. This is an often-hidden performance problem that's worth paying attention to.
- Commonly used APM tools: Splunk, New Relic, Elastic Search

Check Lists, Guidelines and Templates

Linear Development Method of SDLC:

The link below will provide you with

- Method Overview
- SDLC Phases
- Artifacts
- Roles
- Guidelines
- Templates

https://capgemini.sharepoint.com/sites/QAIMS/SEPG/QMS_Wrapper/SitePages/LDM%20Wrapper.aspx

Agile Delivery Lifecycle – Cap Gemini Agile Framework (CAF)

Capgemini Agile Framework (CAF) provides guidance and good practices for efficient execution of our agile engagements. It is lightweight, based on Scrum, extended with eXtreme Programming (XP) practices and complemented by Capgemini's best practices. It can be tailored based on the client needs. It covers activities to be performed for delivering an Agile engagement.

The Link below will provide you with:

- Agile terminologies
- Agile Ceremonies
- Artifacts

- Agile Roles
- Guidelines

https://deliver2.capgemini.com/components/CAF-V4.1/#Capgemini%20Agile%20Framework/guidances/supportingmaterials/CAF%20Overview_B368AD2D.html

Appendix A:

Java-Microservices

Architecture-Context

- https://microservices.io/patterns/microservices.html

Design Patterns

- https://microservices.io/patterns/index.html

Microservices security patterns

- https://tsh.io/blog/microservices-security-patterns/

Dev Ops

Git Hub Commands

- https://git-scm.com/docs/gittutorial

CI/CD Pipeline Set up for Java Apps

- https://buddy.works/docs/quickstart/java

Jenkins CI/CD pipeline setup

https://www.jenkins.io/doc/tutorials/#pipeline

Build a Java app with Maven

- https://www.jenkins.io/doc/tutorials/build-a-java-app-with-maven/

Java

Java 8 Features

- https://www.digitalocean.com/community/tutorials/java-8-features-with-examples

Java Design Solid Principles

- https://www.jrebel.com/blog/solid-principles-in-java

Java OOPS Design Patterns

- https://www.oodesign.com/

Code cleans up Tools for Java Apps

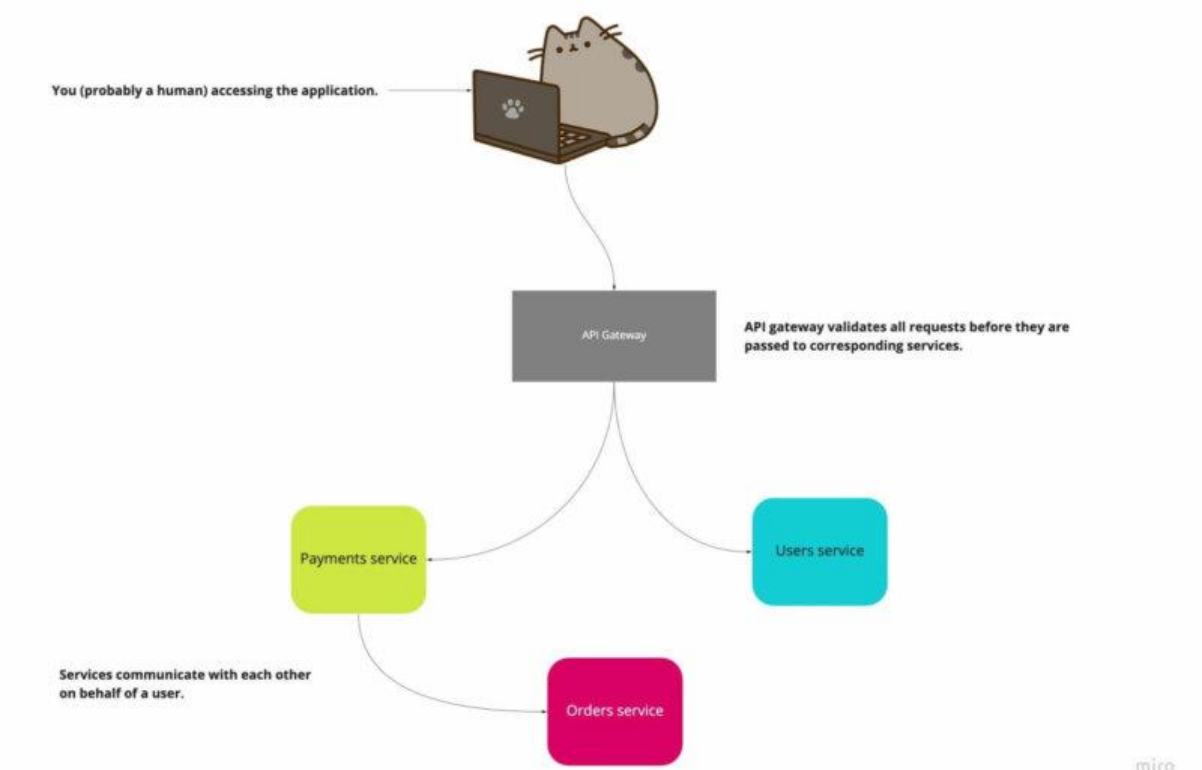
- https://snyk.io/learn/code-review/java-tools/

Mockito, Junit Test cases for Spring boot

- https://howtodoinjava.com/spring-boot2/testing/spring-boot-mockito-junit-example/

Use API (Application Program Interface) Gateway

Usually, microservices applications contain many services that are accessible by different clients and systems. It exposes them to many security risks since it's hard to monitor every single service.



Microservices architecture simplified

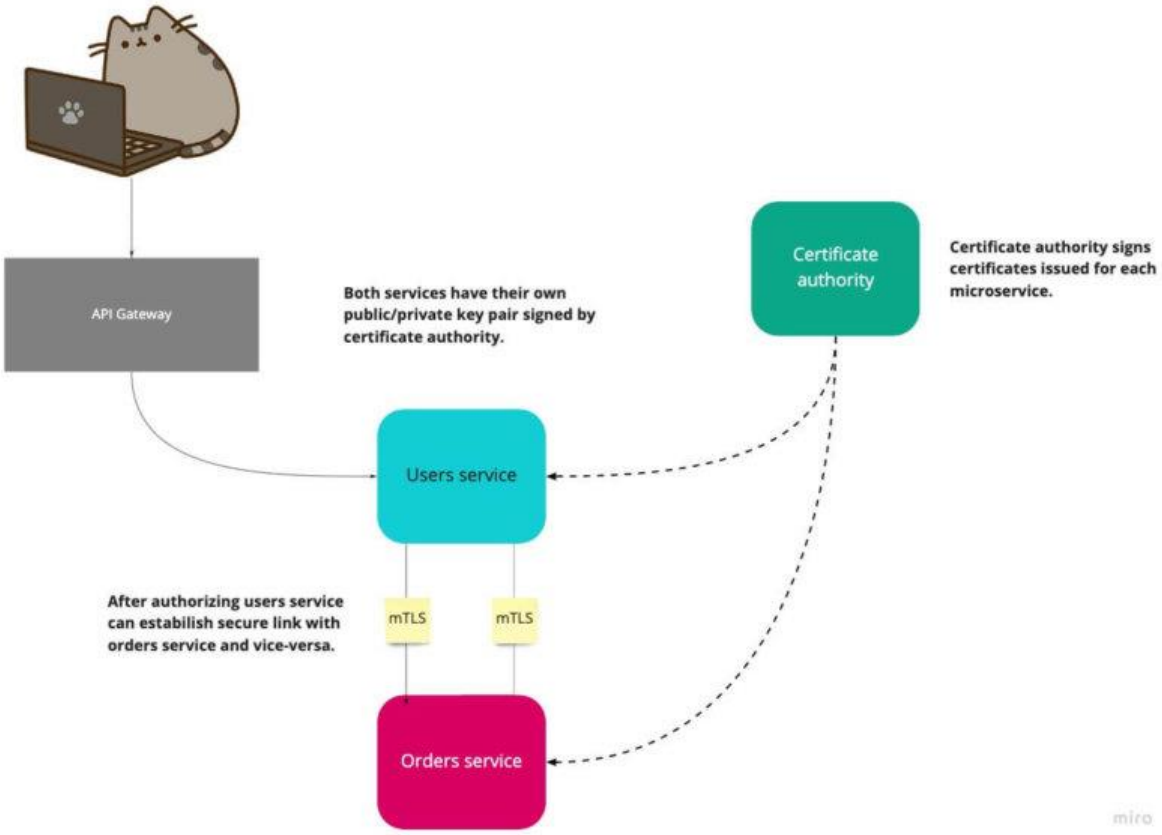
That's why you can deploy API Gateways that handle, monitor and scan all incoming traffic before it is passed to the target service. It will act as a point of entry for all requests

Secure your communication!

Secure communication in microservices is a very important topic. Various testing methods are used such as dynamic application security testing, when the tester tries to enter the app through the frontend. Still, there are some more basic things you can do to implement security in your communication.
By default, you should enforce the use of HTTPS (hypertext transfer protocol secure) in your entire application. When sending sensitive information, such as client credentials, passwords, keys or secrets, encrypt it as soon as possible, and decrypt it as late as possible. Never send this kind of information in plain text.

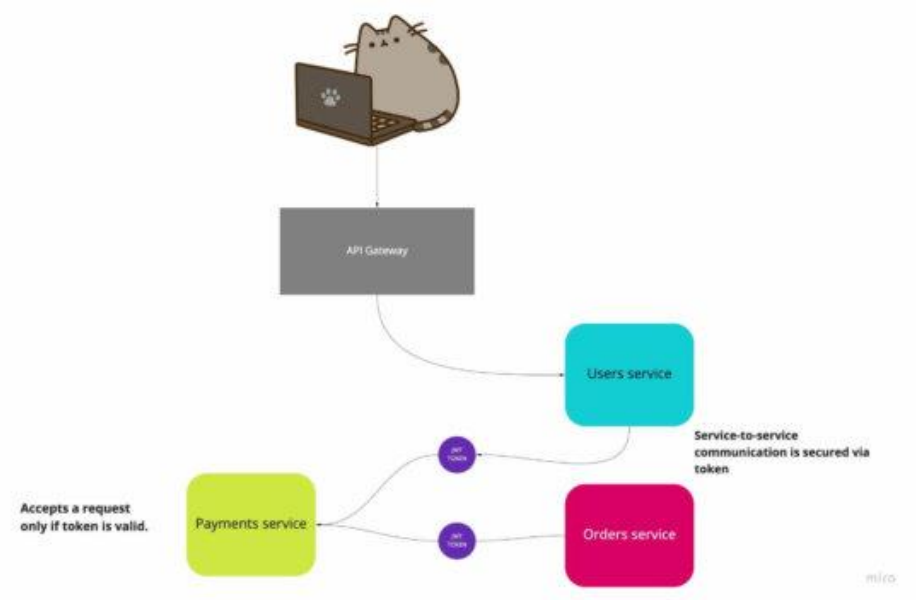
There are several ways how you can secure communication between services. However, these two are the most common: on: Mutual Transport Layer Security (mTLS) and Json Web Token (JWT).

- mTLS – each microservice has a public/private key pair issued by a trusted certificate authority. The client then uses the key-pair to perform basic authentication through the mTLS.



mTLS communication

JWT: each microservice has its own unique JWT token. Clients must know this token to access it (access tokens).



JWT Microservices security pattern for secure communication