Kresil - Kotlin Resilience

Kotlin Multiplatform Library for Fault-Tolerance

Francisco José Barbosa Engenheiro

Supervisor: Pedro Félix

Final report written for Project and Seminary
BSc in Computer Science and Engineering

July 2024

Instituto Superior de Engenharia de Lisboa

Kresil - Kotlin Resilience

Kotlin Multiplatform Library for Fault-Tolerance

49428 Francisco José Barbosa Engenheiro

Supervisor: Pedro Félix, ISEL

Final report written for Project and Seminary
BSc in Computer Science and Engineering

July 2024

Abstract

Text of the abstract. Brief description of the project, important results, and conclusions: the goal is to provide the reader with an overview of the project (should not exceed one page).

Keywords: list of keywords separated by ;.

Resumo

Texto do resumo. Breve descrição do projeto, dos resultados importantes e das conclusões: o objetivo é dar ao leitor uma visão global do projeto (não deve exceder uma página).

Palavras-chave: lista de palavras-chave separadas por ;.

Contents

Chapter 1

Introduction

1.1 Context

In the modern era, our reliance on digital services has grown exponentially, driving the need for these services to be highly reliable and available at all times. Whether it's financial transactions, healthcare systems, or social media platforms, users expect uninterrupted access and seamless experiences. This expectation places significant pressure on the underlying infrastructure to handle failures gracefully and maintain service continuity. Achieving this level of reliability requires sophisticated mechanisms to manage and mitigate faults effectively.

Most of these services are built on distributed systems, which consist of independent networked computers that present themselves to users as a single, coherent system [1]. Given the complexity of these systems, they are susceptible to failures caused by a variety of factors, such as hardware malfunctions, software bugs, network issues, communication problems, or even human errors. As such, its is crucial to ensure that services within distributed systems are resilient, and more specifically, fault-tolerant.

Fault tolerance and fault resilience are key concepts in this context, and while they are related and sometimes used interchangeably, they have subtle differences:

- **Fault Tolerance**: A fault-tolerant service is a service that is able to maintain all or part of its functionality, or provide an alternative, when one or more of its associated components fail. The user does not observe see any fault except for some possible delay during which failover occurs.
- **Fault Resilience**: A fault-resilient service acknowledges faults but ensures that they do not impact committed data (i.e., the database may respond with an error to the attempt to commit a transaction, etc.).

These distinctions are important, because it is possible to regard a fault-tolerant service as suffering *no* downtime even if the machine it is running on crashes, whereas the potential data fault in a fault resilient service counts toward downtime [2].

1.2 Resilience Mechanisms

Over the years, several resilience mechanisms have been developed to help implemented build more robust and reliable systems. These mechanisms provide a set of tools and strategies to handle the

inevitable occurrence of failures. Some of the most common mechanisms are described in table 1.1.

Table 1.1: Resilience mechanisms examples. *Resilience4j* [3] documentation

Name	**Funcionality**	**Description**
Retry	Repeats failed executions.	Many faults are transient and may self-correct after a short delay.
Circuit Breaker	Temporary blocks possible failures.	When a system is seriously struggling, failing fast is better than making clients wait.
Rate Limiter	Limits executions/period.	Limit the rate of incoming requests.
Time Limiter	Limits duration of execution.	Beyond a certain wait interval, a successful result is unlikely.
Bulkhead	Limits concurrent executions.	Resources are isolated into pools so that if one fails, the others will continue working.
Cache	Memorizes a successful result.	Some proportion of requests may be similar.
Fallback	Defines an alternative value to be returned (or action to be executed) on failure.	Things will still fail - plan what you will do when that happens.

These mechanisms can be further categorized based on when they are activated:

- **Reactive Resilience**: Reacts to failures and mitigates their impact (e.g., the *Retry* mechanism is only triggered after a failure occurs);
- **Proactive Resilience**: Prevents failures from happening (e.g., the *Rate Limiter* mechanism is used to limit the rate of incoming requests, as a way to prevent the system from being overwhelmed and potentially fail - acts before a failure occurs).

1.3 Technologies

The main technology used in this project is Kotlin Multiplatform (KMP) [4]. This relatively new technology allows developers to share code across multiple platforms, such as Android and iOS for mobile applications, and/or JVM, JavaScript and Native for multiplatform overall.

The decision to use Kotlin Multiplatform, and more specifically, the Kotlin language, was based on it being the main language used in the B.Sc. in Computer Science and Engineering at Instituto Superior de Engenharia de Lisboa (ISEL) course and the rise in popularity of Kotlin in the industry.

Kotlin is a cross-platform, statically typed, general-purpose high-level programming language with type inference developed by JetBrains, which is fully interoperable with Java [5]. It was designed to have a strong focus on null safety, functional and asynchronous programming, while maintaining the rich feature set of object-oriented programming languages such as Java. Is in constant evolution, with new features and improvements being added regularly [6].

Google announced Kotlin as the official language for Android development in 2019 [5], and more recently, official support for Android development with Kotlin Multiplatform [7, 8].

1.4 Project Goal

The goal of this project is to develop a Kotlin Multiplatform library that provides some of the aforementioned resilience mechanisms and allow for further integration with other libraries and frameworks.

By providing access to these mechanisms in a multiplaform context, developers can integrate them into their projects, regardless of the platform they are targeting.

1.5 Related Work

1.5.1 Ktor

Ktor [9] is a Kotlin Multiplatform framework designed for building asynchronous servers and clients, such as web applications and microservices.

The framework already provides some of the aforementioned resilience mechanisms as plugins, that can installed in the underlying pipepile to intercept specific phases of the request/response cycle and apply the desired behavior (e.g., retrying a request in the client side [10], rate limiting the incoming requests in the server side [11]).

1.5.2 Other Solutions

Traditional Libraries

There are several libraries that provide resilience mechanisms for different programming languages and platforms. The table 1.2 shows some examples of these libraries.

Table 1.2: Examples of libraries that provide resilience mechanisms.

Library	**Language**	**Plataform**
Netflix's Hystrix [12]	Java	JVM
Resilience4j [3]	Java/Kotlin	JVM
Polly [13]	C#	.NET

Hystrix served as an inspiration for Resilience4J, which is based on functional programming concepts. The primary distinction between the two is that, whereas Resilience4J relies on function composition to let you stack the specific decorators you need by utilizing Java 8's features (e.g., functional interfaces, lambda expressions) [14], Hystrix embraces an object-oriented design where calls to external systems have to be wrapped in a *HystrixCommand* offering multiple functionalities.

Resilience4j served as the main source of inspiration for the project's development since it is a more modern way of implementing these mechanisms, follows a functional programming style, and is more in line with the characteristics of the Kotlin language. Polly was used as secondary source to explore alternative approaches and design patterns that could be used in the project.

Arrow Library

The Arrow library, which presents itself as the functional companion to Kotlin's standard library, focuses on functional programming and includes, among other modules, a resilience library. This library implements three of the most critical design pattern around resilience [15]: retry and repeat

computations using a *Schedule*, protect other services from being overloaded using a *CircuitBreaker*, and implement transactional behavior in distributed systems in the form of a *Saga*.

1.6 Document Structure

This document is structured as follows:

- **Chapter 2 - Kotlin Multiplatform**: This chapter provides an overview of Kotlin Multiplatform, its architecture, and how it can be used to share code across multiple platforms.
- **Chapter 3 - Common Design and Implementation Strategy**: This chapter describes the design and implementation aspects that are common to all the resilience mechanisms.
- **Chapter 4 - Retry**: This chapter describes the Retry mechanism, its configuration, and how it was implemented.
- **Chapter 5 - Circuit Breaker**: This chapter describes the Circuit Breaker mechanism, its configuration, and how it was implemented.
- **Chapter 6 - Final Remarks**: This chapter provides a summary of the project in a conclusion format and provides future work considerations.

Chapter 2

Kotlin Multiplatform

The Kotlin Multiplatform (KMP) technology allows developers to share code across multiple platforms, such as Android and iOS for mobile applications, and/or JVM, JavaScript and Native for multiplatform overall.

2.1 Project Structure

A Kotlin Multiplatform project is divided into three main categories of code:

- **Common**: Code shared between all platforms (i.e., *CommonMain, CommonTest*);
- **Intermediate**: Code that can be shared on a subset of platforms (i.e., *AppleMain, AppleTest*);
- **Specific**: Code specific to a target platform (i.e., *<Platform>Main, <Platform>Test*).

An example of a Kotlin Multiplatform project architecture can be seen in Figure 2.1, but note that both *Intermediate* and *Specific* categories are optional.

2.1.1 Template

Although it is possible to create a *KMP* project from scratch, it is recommended to use a template that provides a standardized setup for developing Kotlin libraries and applications that target multiple platforms. This project used the official template available from Kotlin's GitHub organization [16].

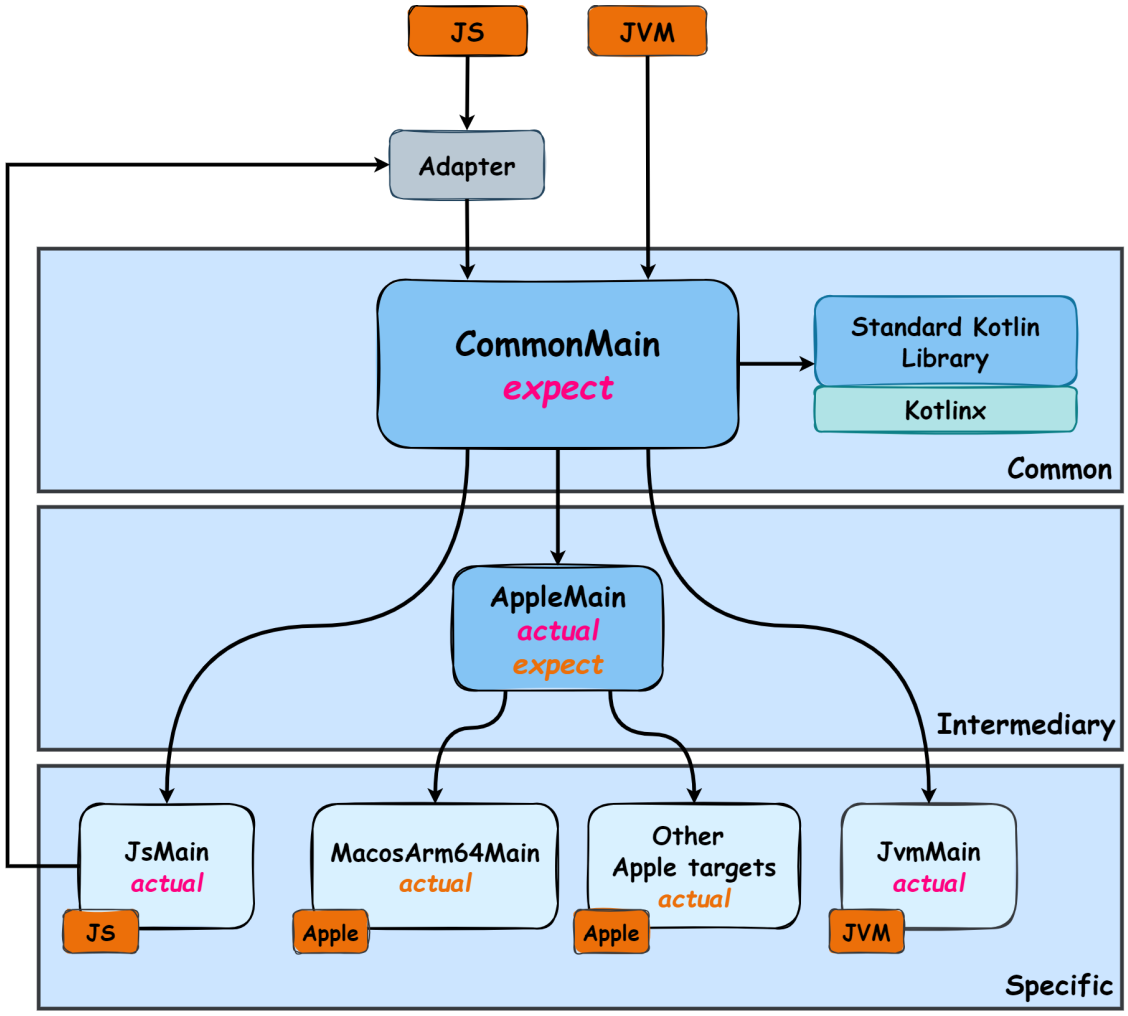
2.1.2 Gradle Tasks

Gradle is a build automation tool for multi-language software development. Offers support for all phases of a build process including compilation, verification, dependency resolving, test execution, source code generation, packaging and publishing [17].

In a Gradle build using Kotlin DSL (i.e., domain-specific language), the project's configuration is primarily defined in two key files:

- *build.gradle.kts* - Defines the project's build configuration. In Kotlin Multiplatform projects, this file is used to define the project's targets, dependencies in the respective source sets, and additional configurations if needed.
- *settings.gradle.kts* - Defines the project's structure and modules it contains.

Figure 2.1: Example of a *KMP* project architecture.



In Gradle, tasks are the smallest unit of work that can be executed and are used to perform specific actions. The templated uses the Kotlin Multiplatform plugin which includes several pre-configured Gradle tasks to facilitate building, testing, and managing the project across multiple platforms configured as targets in the respective build gradle file. Some of the key tasks are:

- **Build**: Compiles and assembles the project;
- **AllTests**: Runs the test cases for all platforms. To run platform-specific tests, use the <Platform>Test task (e.g., *jvmTest*);
- **Check**: Performs various checks on the project, including running tests and performing additional operations (e.g., linting, code analysis);
- **Clean**: Deletes the build directory, allowing for a clean build (i.e., not using cached artifacts).

A Gradle project can be organized into multiple subprojects, each with its own build file, settings and tasks.

2.1.3 GitHub Actions

Provided by the template, the project utilizes GitHub Actions [18] for continuous integration and continuous deployment (CI/CD) [19]. The configurations for GitHub Actions are located in the *.github/workflows* folder, and include the following workflows:

- *gradle.yml* - Builds and tests the project using Gradle against some of the available platforms (e.g., Javascript was missing and was added manually). Runs on push and pull request events to the default branch.
- *deploy.yml* - deploys the library artifacts to a repository in Maven Central, following a pre-defined authentication configuration.

2.1.4 Folder Structure

The project is organized into several folders, each serving a specific purpose. Below is a brief description of each folder, with an emphasis on the *src* folders:

- **.github/**: Contains configurations for GitHub Actions, which are used to automate tasks. See Section 2.1.3 for more information;
- **gradle/**: Contains configuration files and scripts related to the Gradle build system. This folder typically includes:
  - **wrapper/**: Contains the wrapper files and configurations, which standardizes a project on a given Gradle version for more reliable and robust builds [20];
  - **libs.versions.toml**: Defines the versions of the libraries and plugins used in the project, as dependencies, in a centralized manner (regularly known as version catalog).
- **convention-plugins/**: Encapsulates and reuses common build logic across multiple Gradle projects or modules (e.g., for publishing, testing, etc.);
- **library/**: Contains the source code for the library and the build configuration file.
  - **src/**: Contains the source code for the library, divided into multiple submodules based on the target platforms:
    - **commonMain/**: Contains the common code shared across all platforms;
    - **jvmMain/**: Contains the source code specific to the JVM platform;
    - **jsMain/**: Contains the source code specific to the JavaScript platform;
    - **iosMain/**: Contains the source code specific to the iOS platform;
    - **androidMain/**: Contains the source code specific to the Android platform;
    - And all of these module counterparts for the test code (e.g., *commonTest, jvmTest, jsTest, etc*).
  - **build.gradle.kts**: Defines the targets, dependencies, and additional configurations for the library.

- **build.gradle.kts**: The main build configuration file for the project, where the project's targets, dependencies, and additional configurations are defined.
- **settings.gradle.kts**: Configures the Gradle build settings for the project, including the root project name and module inclusion.

Based on the described template's project structure, the following project structure was adopted for developing a Kotlin Multiplatform library:

- **<kmp_package_name>**: name of the Kotlin Multiplatform library in root directory;
  - **apps**: defines the modules that will consume the KMP library (e.g., js-app, android-app)
  - **lib/shared**: defines the library's code to be shared by the consuming modules;
    - **src** defines the target submodules of the library including their test counterparts (i.e., <Platform>Main, <Platform>Test);
    - **build.gradle.kts**: defines the library's dependencies, targets, and additional configurations;

2.2 Platform-Dependent Code

As code sharing across platforms is the primary objective of *KMP*, the code should be written as platform-independently as possible (i.e., aggregating as much code as possible in the hierarchically higher categories). However, it is sometimes necessary to create specific code for a given platform, regularly referred to as *target*, in the following situations:

- Access to API's specific to the *target* is required (e.g., *Java's File API*);
- The libraries available in the common category (i.e., *Standard Kotlin Library*, libraries from *Kotlinx*) do not cover the desired functionalities and third-party libraries either don't support it or dependency reduction is desired;
- A given *target* does not directly support *KMP* (e.g., *Node.js)*, and so it is necessary to create an *adapter*. This adapter allows communication with the common category code, in *Kotlin*, from the native code of the *target*, which can be defined in the *Intermediate* or *Specific* category.

To create specific code for a *target*, the mechanism ***expect/actual*** [21] is used. This mechanism allows defining the code to be implemented in an abstracted way and its implementation, respectively.

2.3 Supported Targets

The project supports the following targets:

- **Jvm**: Allows running the code on the Java Virtual Machine;
- **JavaScript**: Allows running the code in a browser or Node.js environment;

- **Android**: Allows running the code on Android devices.
- **Native**: Allows running the code on platforms that support Kotlin/Native, excluding macOS and iOS, because the lack of access to the necessary hardware for testing.

2.4 Running Tests

TODO

2.5 Other Aspects

TODO: What was done to have concurrency, logging, CI integration, etc

Chapter 3

Common Design and Implementation Strategy

3.1 Mechanism Model

After a thorough research and analysis of the available mechanisms implementations, it was identified the common design aspects that are shared among them. These aspects, turned components, form the foundation of the future mechanisms implementations and are represented in the Mechanism Model, as shown in Figure 3.1.

The Mechanism Model is composed of the following components:

- **Configuration**: Represents a set of policies that, in conjuction, define the mechanism's behavior (e.g., maximum number of retries, maximum wait duration, etc.);
- **Asynchronous Context**: Represents the mechanism's execution context, responsible for state management and event emission;
- **State**: Represents the internal state of the mechanism;
- **Implementation**: Applies the configuration to the mechanism's execution context. Represents the core component of the mechanism;
- **Registry**: Acts as a centralized container for storing and managing available mechanism implementations and their configurations. The registry allows access to mechanism implementations throughout the application and enables the reuse of configurations to create new mechanisms.
- **Events**: Both the Asynchronous Context and Registry components are responsible for emitting events. The Asynchronous Context component emits events related to the mechanism's execution, such as internal state transition changes. The Registry component emits events related to CRUD operations (Create, Read, Update, Delete) performed in the registry. These events can be used for various purposes, such as logging and monitoring.
- **Metrics**: The mechanism's implementation component is responsible for recording metrics related to the mechanism's execution (e.g., number of retries, number of recorded failures, etc.). These metrics can be used for monitoring and analysis purposes.

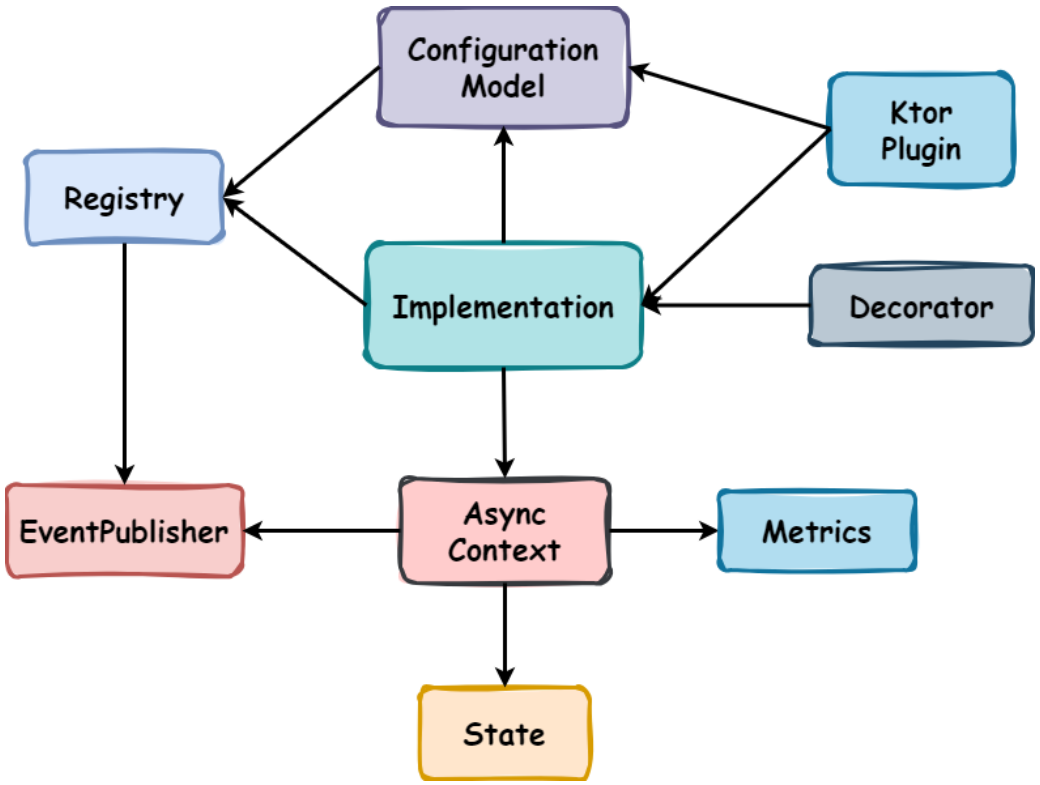


Figure 3.1: Mechanism Model

- **Decorator**: The decorator is an extension of the Implementation component. It is based on Resilience4J [3] decorators (i.e., high-order functions), and provide a convenient way to wrap code blocks with the mechanism's behavior.

- **Ktor Plugin**: Responsible for the integration of the mechanism implementation with the Ktor pipeline. The Configuration component is also used to create a specific plugin configuration, which can be used to extend the mechanism's behavior and provide additional features in an HTTP context.

3.1.1 Configuration Design

Since the start of the project, the Configuration component was designed to use the builder pattern [22]. This way, separating the configuration definition (mutable) from the configuration usage (immutable). However, the initial implementation had a limitation: it was not possible to override a configuration object (i.e., create a new configuration object based on an existing one and only change a few properties while keeping the rest).

In order to overcome this limitation, the Configuration component, and more specifically, its builder, was redesigned to always receive a base configuration object in its creation. This modification allows for incremental configuration, essentially following the pattern: `config(default/initial) -> configBuilder -> config -> configBuilder -> config -> (...)` A.

3.1.2 Mechanism Execution Context

The mechanism's execution context is crucial for ensuring proper operation across synchronous and asynchronous environments. Since the mechanisms are designed to be cross-platform, the execution context must be flexible and support asynchronous operations, particularly for JavaScript, one of the supported targets (see Section 2.3), which is single-threaded and requires asynchronous (non-blocking) operations.

Independent of the execution environment, the execution context is one of the following forms:

- **Per Mechanism**: A new execution context is created when the mechanism itself is instantiated (e.g., the Circuit Breaker mechanism has a single execution context for managing the circuit state, as multiple callers can interact with the mechanism at the same time).
- **Per Decoration**: When a decorator is applied to an operation, it creates a new execution context specific to that decoration, before invoking the underlying operation.
- **Per Method Invocation**: A new execution context is created each time the decorated method is invoked. This is the most granular form of execution context, providing isolation for each method call. If the underlying operation is thread-safe, then this form of execution context is also thread-safe, as only the calling thread executes the context (e.g., the Retry mechanism creates a new execution context for each underlying operation invocation).

3.2 Ktor Integration

Integration with Ktor was considered from the beginning of the project, because Ktor:

- provides a flexible pipeline-based architecture, which allows for integration of custom behavior;
- is an official JetBrains product;
- is also a multiplatform framework, although server-side is limited to the JVM and Native targets [23].

Additionally, integrating with another library or framework (beyond just Ktor) provided a real-world use case for the implemented mechanisms and presented a development exercise. The challenge was not only integrate the library with a third-party framework, but also ensure that the implemented mechanisms are well-equipped to handle different scenarios and environments without being dependent on a specific context (e.g., HTTP); and also provide extension points for additional features and customizations that only make sense in a specific context (e.g., retrying on server responses with specific status codes).

3.2.1 Plugin

In Ktor, a plugin is a reusable component that can be installed in an application to extend its functionality. They represent a way to encapsulate common functionality (e.g., logging, authentication, serialization, etc.) and make it reusable across different applications. Plugins are installed in the

application's pipeline, where they can intercept and modify the request and response processing flow, as the Figure 3.2 demonstrates for the server side.

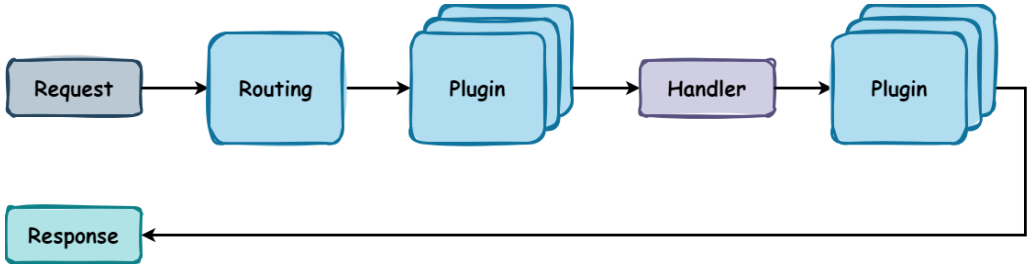


Figure 3.2: Ktor Server Architecture

In the server side, as a request comes in, it goes through a series of steps [24]:

1. It is routed to the correct handler via the routing mechanism (which is also a plugin);
2. Before being handed off to the handler, it goes through one or more Plugins;
3. The handler (application logic) handles the request;
4. Before the response is sent to the client, it goes through one or more Plugins

3.2.2 Pipeline

A Pipeline, represented in Figure 3.3, is a collection of zero or more interceptors, grouped in one or more ordered phases. Each interceptor can perform custom logic before and after processing a request.

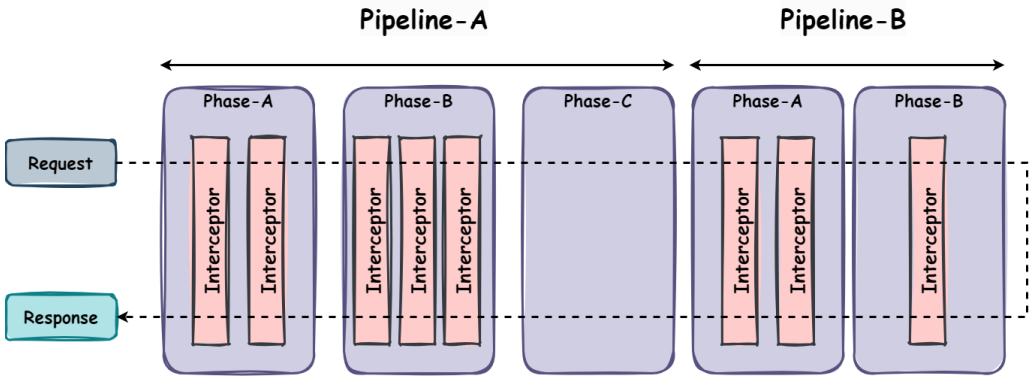


Figure 3.3: Ktor Pipeline Example

A plugin is also an interceptor, but an interceptor is not a plugin. While an interceptor is a function block that can be added to intercept a specific pipeline phase and perform custom logic; a plugin is a collection of zero or more interceptors mixed with external configuration and other logic (e.g., add more pipeline phases) in a single reusable component.

Both client and server sides have pipelines, but they differ in terms of their phases and purposes. Tables 3.1 and 3.2 show the phases of the server and client pipelines, respectively.

Table 3.1: Ktor Server Pipelines

Pipeline	Description	Phases
ApplicationSend	Responsible for sending responses	Before, Transform, Render, Content-Encoding, Transfer-Encoding, After, Engine
ApplicationReceive	Responsible for receiving requests	Before, Transform, After
ApplicationCall	Responsible for executing application calls	Setup, Monitoring, Plugins, Call, Fallback

Table 3.2: Ktor Client Pipelines

Pipeline	Description	Phases
HttpRequest	Processes all requests sent by the client	Before, State, Transform, Render, Send
HttpSend	Used for send a request	Before, State, Monitoring, Engine, Receive
HttpReceive	Used for receiving a response without processing	Before, State, After
HttpResponse	Used for processing responses	Receive, Parse, Transform, State, After

Client Pipelines

3.2.3 Custom Plugins

Ktor provides a custom plugin API that allows developers to create their own plugins in both client and server sides.

Since Ktor 2.0.0, the custom plugin API has been simplified [25, 26] and no longer requires an understanding of internal Ktor concepts, such as pipelines, phases, etc. Instead, developers have access to different stages of handling requests and responses using general handlers (e.g., `onCallReceive`, and `onCallRespond`), which intercept the related phases of the pipeline.

3.3 Development Roadmap

Mechanism Model Implementation for a specific mechanism -¿ Tests -¿ All targets support -¿ Ktor plugin and repeat for each mechanism in a vertical manner.

Chapter 4

Retry

4.1 Introduction

- Why it exists (1) - Functional characterization (2)

4.2 Configuration

- mention default values and why they were chosen

4.3 Implementation Aspects

4.4 Ktor Integration

Chapter 5

Circuit Breaker

5.1 Introduction

- Why it exists (1) - Functional characterization (2)

5.2 Configuration

- mention default values and why they were chosen

5.3 Implementation Aspects

5.4 Ktor Integration

Bibliography

[1] FreeCodeCamp contributors. A thorough introduction to distributed systems. https://www.freecodecamp.org/news/a-thorough-introduction-to-distributed-systems-3b91562c9b3c, 2024. [Online; accessed 5-March-2024].

[2] James Bottomley. Fault tolerance vs fault resilience. https://www.usenix.org/legacy/publications/library/proceedings/usenix04/tech/sigs/full_papers/bottomley/bottomley_html/node21.html, 2004. [Online; accessed 21-May-2024].

[3] Resilience4j contributors. Resilience4j: User guide. https://resilience4j.readme.io/docs/getting-started, 2024. [Online; accessed 6-March-2024].

[4] JetBrains contributors. Kotlin multiplatform. https://kotlinlang.org/docs/multiplatform.html, 2024. [Online; accessed 7-March-2024].

[5] Wikipedia contributors. Kotlin (programming language). https://en.wikipedia.org/wiki/Kotlin_(programming_language), 2024. [Online; accessed 22-May-2024].

[6] JetBrains contributors. Kotlin evolution and enhancement process. https://github.com/Kotlin/KEEP, 2024. [Online; accessed 22-May-2024].

[7] Google Developers. Android support for kotlin multiplatform to share business logic across mobile, web, server, and desktop. https://android-developers.googleblog.com/2024/05/android-support-for-kotlin-multiplatform-to-share-business-logic-across-mobile-web-server.html, 2024. [Online; accessed 22-May-2024].

[8] Touchlab. Google i/o 2024: Kotlin multiplatform at google scale! https://touchlab.co/KMP-at-google, 2024. [Online; accessed 22-May-2024].

[9] JetBrains contributors. Ktor: Web applications. https://ktor.io, 2024. [Online; accessed 7-March-2024].

[10] Ktor Contributors. Ktor: Client request retry plugin. https://ktor.io/docs/client-request-retry.html, 2024. [Online; accessed 22-May-2024].

[11] Ktor Contributors. Ktor: Server rate limit plugin. https://ktor.io/docs/server-rate-limit.html, 2024. [Online; accessed 22-May-2024].

[12] Netflix contributors. Hystrix: Latency and fault tolerance for distributed systems. https://github.com/Netflix/Hystrix, 2024. [Online; accessed 6-March-2024].

[13] App-vNext contributors. Polly: Resilience strategies. https://github.com/App-vNext/Polly#resilience-strategies, 2024. [Online; accessed 6-March-2024].

[14] Exoscale. Migrate from hystrix to resilience4j. https://www.exoscale.com/syslog/migrate-from-hystrix-to-resilience4j/, 2024. [Online; accessed 22-May-2024].

[15] Arrow Contributors. Introduction to resilience. https://arrow-kt.io/learn/resilience/intro/, 2024. [Online; accessed 22-May-2024].

[16] JetBrains contributors. Kotlin multiplatform: Github template. https://github.com/Kotlin/multiplatform-library-template, 2024. [Online; accessed 22-May-2024].

[17] Wikipedia contributors. Gradle. https://en.wikipedia.org/wiki/Gradle, 2024. [Online; accessed 22-May-2024].

[18] GitHub Contributors. Github actions. https://github.com/features/actions, 2024. [Online; accessed 22-May-2024].

[19] Red Hat Contributors. What is CI/CD? https://www.redhat.com/en/topics/devops/what-is-ci-cd, 2024. [Online; accessed 22-May-2024].

[20] Gradle Contributors. Gradle wrapper. https://docs.gradle.org/current/userguide/gradle_wrapper.html, 2024. [Online; accessed 22-May-2024].

[21] JetBrains contributors. Kotlin multiplatform: Expect/actual. https://kotlinlang.org/docs/multiplatform-expect-actual.html, 2024. [Online; accessed 12-March-2024].

[22] Wikipedia contributors. Builder pattern. https://en.wikipedia.org/wiki/Builder_pattern, 2024. [Online; accessed 23-May-2024].

[23] Ktor Contributors. Ktor server platforms. https://ktor.io/docs/server-platforms.html, 2024. [Online; accessed 23-May-2024].

[24] Ktor Contributors. Adding functionality with server plugins. https://ktor.io/docs/server-plugins.html#add_functionality, 2024. [Online; accessed 23-May-2024].

[25] Ktor Contributors. Ktor server custom plugins. https://ktor.io/docs/server-custom-plugins.html, 2024. [Online; accessed 24-May-2024].

[26] Ktor Contributors. Ktor client custom plugins. https://ktor.io/docs/client-custom-plugins.html, 2024. [Online; accessed 24-May-2024].

Appendix A

Config Builder Interface

Etiam ac leo a risus tristique nonummy. Donec dignissim tincidunt nulla. Vestibulum rhoncus molestie odio. Sed lobortis, justo et pretium lobortis, mauris turpis condimentum augue, nec ultricies nibh arcu pretium enim. Nunc purus neque, placerat id, imperdiet sed, pellentesque nec, nisl. Vestibulum imperdiet neque non sem accumsan laoreet. In hac habitasse platea dictumst. Etiam condimentum facilisis libero. Suspendisse in elit quis nisl aliquam dapibus. Pellentesque auctor sapien. Sed egestas sapien nec lectus. Pellentesque vel dui vel neque bibendum viverra. Aliquam porttitor nisl nec pede. Proin mattis libero vel turpis. Donec rutrum mauris et libero. Proin euismod porta felis. Nam lobortis, metus quis elementum commodo, nunc lectus elementum mauris, eget vulputate ligula tellus eu neque. Vivamus eu dolor.

Nulla in ipsum. Praesent eros nulla, congue vitae, euismod ut, commodo a, wisi. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Aenean nonummy magna non leo. Sed felis erat, ullamcorper in, dictum non, ultricies ut, lectus. Proin vel arcu a odio lobortis euismod. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Proin ut est. Aliquam odio. Pellentesque massa turpis, cursus eu, euismod nec, tempor congue, nulla. Duis viverra gravida mauris. Cras tincidunt. Curabitur eros ligula, varius ut, pulvinar in, cursus faucibus, augue.

Nulla mattis luctus nulla. Duis commodo velit at leo. Aliquam vulputate magna et leo. Nam vestibulum ullamcorper leo. Vestibulum condimentum rutrum mauris. Donec id mauris. Morbi molestie justo et pede. Vivamus eget turpis sed nisl cursus tempor. Curabitur mollis sapien condimentum nunc. In wisi nisl, malesuada at, dignissim sit amet, lobortis in, odio. Aenean consequat arcu a ante. Pellentesque porta elit sit amet orci. Etiam at turpis nec elit ultricies imperdiet. Nulla facilisi. In hac habitasse platea dictumst. Suspendisse viverra aliquam risus. Nullam pede justo, molestie nonummy, scelerisque eu, facilisis vel, arcu.