ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

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 adopt Kotlin Multiplatform, particularly the Kotlin language, stemmed from its prominence as the primary language in the BSc in Computer Science and Engineering program at Instituto Superior de Engenharia de Lisboa (ISEL). Moreover, the course structure, was recently influenced by the widespread adoption and popularity of Kotlin within the software development community.

Kotlin [5] is a cross-platform, statically typed, general-purpose high-level programming language with type inference developed by JetBrains [6], which is fully interoperable with Java. 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 [7].

Google announced Kotlin as the official language for Android development in 2019 [5], and more recently, official support for Android development with Kotlin Multiplatform [8, 9].

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 [10] 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 [11], rate limiting the incoming requests in the server side [12]).

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 [13]	Java	JVM
Resilience4j [3]	Java/Kotlin	JVM
Polly [14]	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) [15], 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 [16]: 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 the Kotlin Multiplatform technology, its architecture, and how it can be used to share code across multiple platforms; Additionally, it describes the project structure, the adopted project structure, and how to run tests in a multiplatform context;

- **Chapter 3 - Common Design and Implementation Strategy**: This chapter describes the design and implementation aspects that are common to all the resilience mechanisms; Additionally, it describes the Ktor framework and how it was used in the project;

- **Chapter 4 - Retry**: This chapter describes the Retry mechanism functionality, its configuration, how it was implemented in both the library and the Ktor framework as a plugin;

- **Chapter 5 - Circuit Breaker**: This chapter describes the Circuit Breaker mechanism functionality, its configuration, how it was implemented in both the library and the Ktor framework as a plugin;

- **Chapter 6 - Final Remarks**: This chapter provides a summary of the project in a conclusion format and ends with 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 [17].

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 [18].

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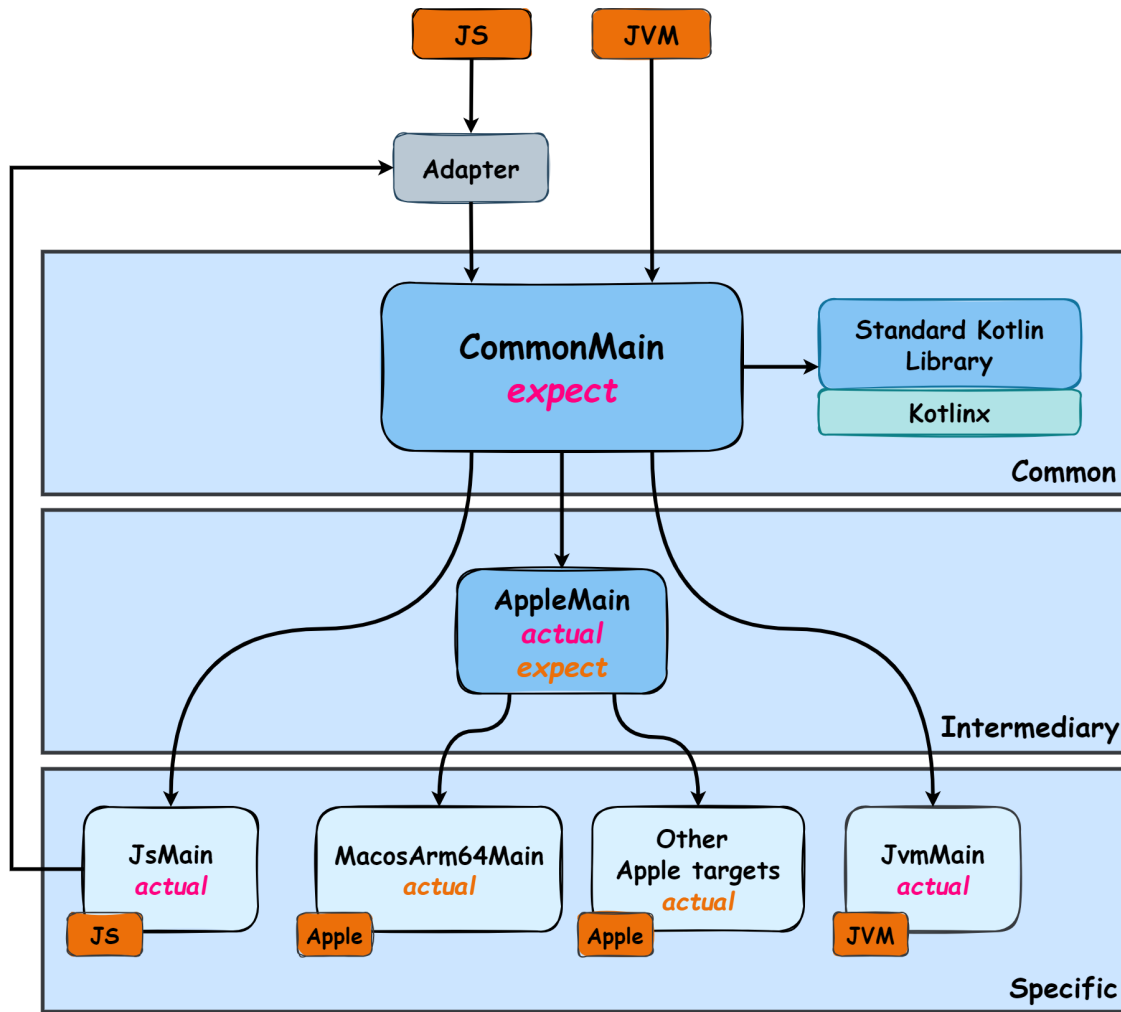
Figure 2.1: Example of a *KMP* project architecture.

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 [19] for continuous integration and continuous deployment (CI/CD) [20]. 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 predefined 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 [21];

  - **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 modules common dependencies 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 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* [22] 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 in an asynchronous environment;

- **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* [23] operations 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;

- **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.
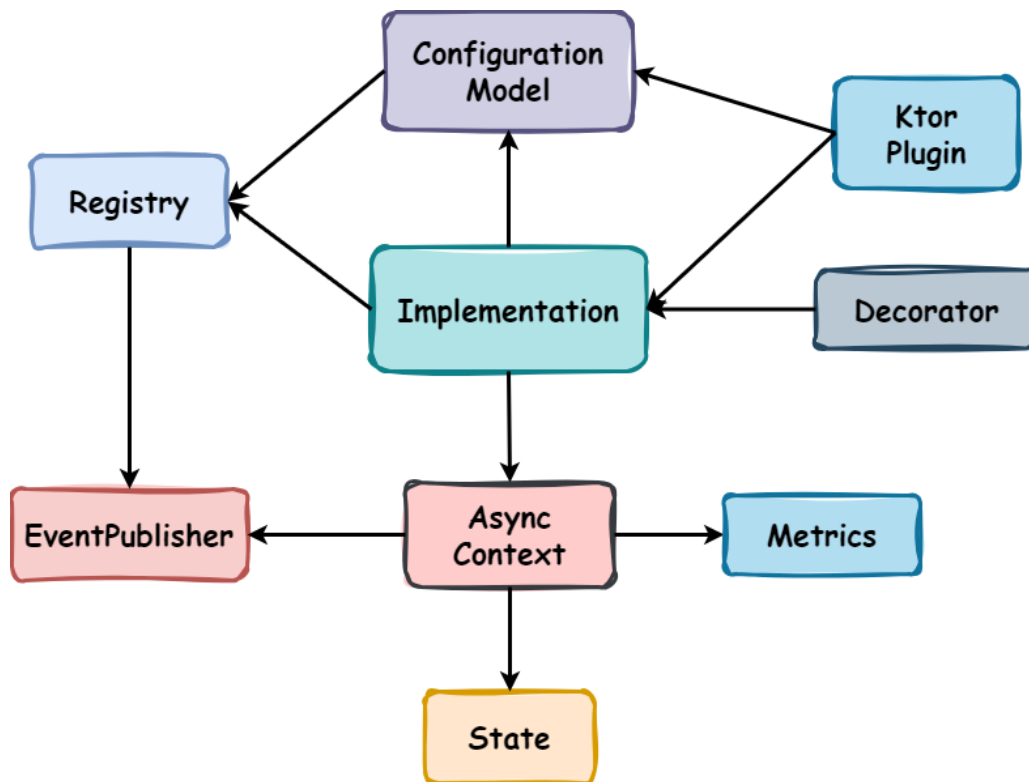

Figure 3.1: Mechanism Model

3.1.1 Configuration Design

Since the start of the project, the Configuration component was designed to use the builder pattern [24], this way, separating the configuration definition (mutable) from the configuration usage (immutable). However, this 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 modify specific properties while keeping the rest of the properties unchanged).

To address this limitation, the Configuration component, particularly its builder, was redesigned to always start with a base configuration object. This redesign enables incremental configuration, following a pattern where a configuration object is continually modified until the desired configuration is achieved.

The configuration process is as follows:

1. Begin with a default or initial (base) configuration object;

2. Pass this configuration to a configuration builder;

3. The builder potentially modifies the base configuration;

4. Generate a new configuration from the builder;

5. If further modifications are needed, pass the new configuration back to the builder, which will use it as the base configuration;

6. Repeat the process until the desired configuration is achieved.

3.1.2 Mechanism Execution Context

Independent of the execution environment, synchronous or asynchronous, the mechanism's execution context can only be one of the following:

- **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).

3.1.3 Asynchronous Context

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.

In Kotlin, asynchronous operations are typically handled using Kotlin Coroutines [25], which provide a way to write asynchronous code in a sequential manner, leveraging the concept of suspending functions.

Thread

In computer science, a thread [26] of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

In addition to being managed independently by an operating system's scheduler, threads are fundamental units of execution within a process [27] They enable concurrent execution of tasks, allowing a program to perform multiple operations simultaneously.

Kotlin Coroutines

A coroutine is an instance of a potentially suspendable computation, which can be suspended and resumed at a later time. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code [25]. However, a coroutine is not bound to any particular thread, as it may suspend its execution in one thread and resume in another one.

A suspended coroutine does not block the underlying thread, freeing it to run other coroutines or perform other tasks. In a thread, only one coroutine can run at a given time.

Another important feature of coroutines is that they enforce structured concurrency [25]. Structured concurrency is a programming paradigm that enforces a hierarchical structure on concurrent code, ensuring that all concurrent tasks are properly managed and cleaned up, and any errors are propagated correctly and not lost.

3.1.4   Event Publishing

Since the execution context is asynchronous, the events must be published asynchronously. To achieve this, the Kotlin Coroutines' asynchronous primitive Flow [28] was used to emit events.

Flow

A Flow [28] is a cold asynchronous stream that sequentially emits values and completes normally or with an exception.

A cold stream starts emitting values only when it is collected, and each collector receives all the values from the beginning; this is in contrast to hot streams, which emit values regardless of whether they are collected or not.

Flows differ from Sequences [29] in that they are used for handling asynchronous streams of data that can emit multiple values over time, while sequences are used for handling synchronous streams of data that are computed on demand (lazy) and can only produce values once.

Implementation

All events in a mechanism are emitted using a hot Flow with no buffer, which means that, if there are no collectors (listeners) for the events, they are essentially lost (not recorded). As such, a listener must be registered before the event is emitted, in order to receive it.

Every mechanism implementation has two ways to register listeners for events:

- **onEvent**: Registers a listener to receive all events emitted by the mechanism;

- **on<SpecificEvent>**: Registers a listener to receive only a specific type of event emitted by the mechanism.

The implementation also provides a way to cancel all registered listeners, by leveraging the coroutine's structured concurrency [25], which is useful for cleaning up resources when the mechanism is no longer needed. The cancellation of the registered listeners up to a given time, does not affect subsequent registrations.

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 [6] product;

- is also a multiplatform framework, although server-side is limited to the JVM and Native targets [30].

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.
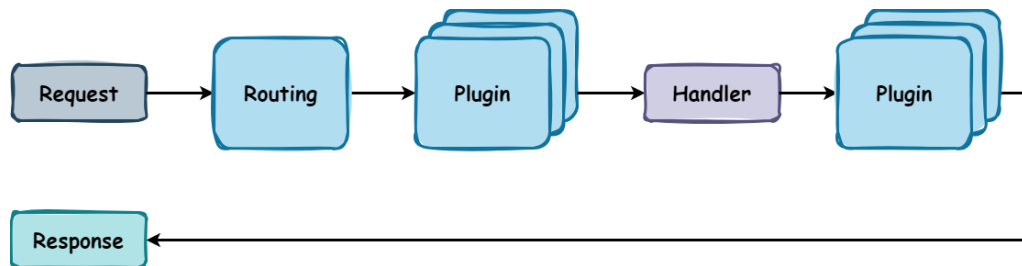

Figure 3.2: Ktor Server Architecture

In the server side, as a request comes in, it goes through a series of steps [31]:

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
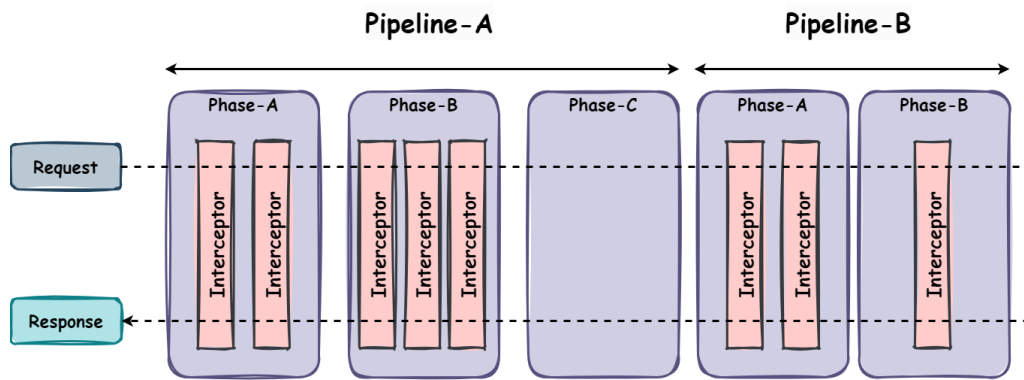
Figure 3.3: Ktor Pipeline Example

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.

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

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 [32, 33] 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

Originally, the project was planned to be developed in a horizontal manner, where all mechanisms would be implemented for all targets at the same time, tested, and then integrated with Ktor.

However, due to the complexity of the mechanisms and the need to ensure that they are correctly implemented and tested in other contexts, the development strategy was changed to a vertical approach.

For each mechanism, the following steps are taken:

1. Implement the Mechanism Model for a specific mechanism, including all of its components;

2. Write tests for the implemented mechanism;

3. Ensure that the implemented mechanism work correctly for all targets;

4. Implement the Ktor Plugin that uses the implemented mechanism;

5. Write tests for the implemented Ktor Plugin; Due to time constraints, unit tests and integration tests were not conducted; however, functional tests [34] were performed using a real Ktor application;

Chapter 4

Retry

4.1 Introduction

The retry mechanism is a resilience pattern that allows an application to handle transient failures when it tries to connect to a service or network resource. By transparently retrying a failed operation, the application can improve its stability and availability [35].

This mechanism is particularly useful when the application is interacting with services that are prone to temporary failures, such as network issues, temporary unavailability of services, or timeouts when services are overloaded. These issues are often brief and resolve themselves within a short period, meaning that retrying the operation after a short delay can often succeed (e.g., a call to a service that is temporarily overloaded might succeed if retried after a few seconds) [35].

Without a retry mechanism, an application might treat such transient failures as critical, leading to unnecessary disruptions in service, increased latency, and a poor user experience.

If an application detects a failure when it tries to send a request to a remote service, it can handle the failure using the following strategies [35]:

- **Cancel**: If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception (e.g., an authentication failure caused by providing invalid credentials is not likely to succeed no matter how many times it's attempted);

- **Retry**: If the specific fault reported is unusual or rare, it might have been caused by unusual circumstances (e.g., a transient network issue). In this case, the application could retry the failing request again immediately because the same failure is unlikely to be repeated (i.e., no-delay retry);

- Retry after delay. If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period while the connectivity issues are corrected or the backlog of work is cleared. The application should wait for a suitable time before retrying the request. The amount of time to wait before retrying, depends on:

  - the type of failure and the probability that it'll be corrected during this time;
  - the delay strategy used (e.g., constant, linear, exponential);

19

4.1.1 Delay Strategies

The retry delay strategy defines the amount of time the application should wait before retrying the operation, and it can be one of the following:

- **No Delay**: This strategy does not introduce any delay between retries;

- **Constant Delay**: Introduces a fixed, constant delay between each retry attempt;

- **Linear Delay**: Increases the delay duration linearly with each retry attempt. The delay is calculated by multiplying the initial delay by the retry attempt number (e.g., initial delay=1s, retry attempt=4, result=[1, 2, 3, 4]);

- **Exponential Delay**: This strategy exponentially increases the delay duration with each retry attempt, by using the exponential backoff algorithm [36]. Essentially, the delay is calculated by multiplying the initial delay by a specified multiplier raised to the power of the retry attempt number (e.g., initial delay=1s, multiplier=2, retry attempt=4, result=[1, 2, 4, 8]);

In both linear and exponential delay strategies, a maximum delay can be set to prevent potentially excessive delays caused by the growth of the delay duration with each retry attempt. Additionally, a *jitter* [37] can be introduced to randomize the calculated delay duration. This randomization can help avoid synchronization between multiple clients that are retrying the same operation at the same time, commonly known as the thundering herd problem [38], by spreading out the retries over a period of time.

4.2 Configuration

- Table with configuration options (do not mention result mapper as it is in experimental phase) - TODO: mention default values and why they were chosen

4.3 Implementation Aspects

- Describe what deviates from the Mechanism model

4.3.1 Execution Flow

- Describe the state machine and the possible transitions, use drawing in presentation

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. Jetbrains. `https://www.jetbrains.com/`, 2024. [Online; accessed 24-May-2024].

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

[8] 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].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[23] Wikipedia contributors. Create, read, update and delete. `https://en.wikipedia.org/wiki/Create,_read,_update_and_delete`, 2024. [Online; accessed 24-May-2024].

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

[25] JetBrains contributors. Kotlin coroutines. `https://kotlinlang.org/docs/coroutines-basics.html`, 2024. [Online; accessed 24-May-2024].

[26] Wikipedia contributors. Thread (computing). `https://en.wikipedia.org/wiki/Thread_(computing)`, 2024. [Online; accessed 24-May-2024].

[27] Wikipedia contributors. Process (computing). `https://en.wikipedia.org/wiki/Process_(computing)`, 2024. [Online; accessed 24-May-2024].

[28] JetBrains contributors. Kotlin flow. `https://kotlinlang.org/docs/flow.html#flows`, 2024. [Online; accessed 24-May-2024].

[29] JetBrains contributors. Kotlin sequences. `https://kotlinlang.org/docs/sequences.html`, 2024. [Online; accessed 24-May-2024].

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

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

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

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

[34] Software Testing Help. The difference between unit, integration, and functional testing. `https://www.softwaretestinghelp.com/the-difference-between-unit-integration-and-functional-testing/`, 2024. [Online; accessed 24-May-2024].

[35] Microsoft. Retry pattern. `https://learn.microsoft.com/en-us/azure/architecture/patterns/retry`, 2023. [Online; accessed 24-May-2024].

[36] Wikipedia contributors. Exponential backoff. `https://en.wikipedia.org/wiki/Exponential_backoff`, 2024. [Online; accessed 24-May-2024].

[37] Wikipedia contributors. Jitter. `https://en.wikipedia.org/wiki/Jitter`, 2024. [Online; accessed 24-May-2024].

[38] Wikipedia contributors. Thundering herd problem. `https://en.wikipedia.org/wiki/Thundering_herd_problem`, 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.