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 always. 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, it 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].

1.1.1 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).

1.1.2 Technologies

Kotlin Multiplatform is a technology that 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. This type of technology is particularly useful for developers who want to build services that run on multiple platforms, without having to write potentially equivalent platform-specific code.

In 2019, Google announced it as the official language for Android development in [4], and more recently, official support for Android development with Kotlin Multiplatform [5, 6]. This recognition strengthens the importance and interest of the technology in the software development community.

As a recent technology that is still evolving, and only reached stable status late last year [7], it lacks the necessary tools (e.g., libraries, frameworks, etc.) that provide the functionalities that are common in other ecosystems.

1.2 Problem

An analysis of the Kotlin Multiplatform ecosystem has revealed a significant gap: there is no specialized library that offers a wide variety of resilience mechanisms. This absence is a critical issue because resilience mechanisms are essential for building reliable and robust systems.

To address this problem, it is crucial to develop a library that enables developers to integrate resilience mechanisms seamlessly into their projects, regardless of the platform they are targeting.

1.3 Related Work

1.3.1 Ktor

Ktor [8] 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 be installed in the underlying pipepile to intercept specific phases of the request/response cycle and apply the desired behaviour (e.g., retrying a request on the client side [9], rate limiting the incoming requests on the server side [10]).

1.3.2 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 [11]	Java	JVM
Resilience4j [3]	Java/Kotlin	JVM
Polly [12]	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) [13], 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 was considered 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 a 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 patterns around resilience [14]: retry and repeat computations using a *Schedule*, protect other services from being overloaded using a *CircuitBreaker*, and implement transactional behaviour in distributed systems in the form of a *Saga*.

1.4 Project Goal

The goal of this project is to develop a dedicated Kotlin Multiplatform library that provides some of the aforementioned resilience mechanisms.

Additionally, the project aims to provide access to these mechanisms through plugins for the Ktor framework. This integration was considered because:

- Ktor is the only known Kotlin Multiplatform framework for server and client HTTP services development;
- Immediately provide a way to use the library in a specific context;
- Validate the implementation and extensibility of the library.

The library should be easy to use and configure, taking advantage of Kotlin's language features to provide a concise and expressive API.

The intended mechanisms to be implemented and the respective plugins for the Ktor framework are:

- Retry and Plugin Mechanism for Ktor Client;
- Circuit Breaker Mechanism and Plugin for Ktor Client;
- Rate Limiter Mechanism and Plugin for Ktor Server;
- Cache Mechanism and Plugin for Ktor Server

The project will also provide a battery of tests to ensure the correctness and reliability of the implemented mechanisms and plugins, as well as documentation to guide developers on how to use them.

1.5 Document Structure

This document is structured as follows:

- **Chapter 2 - Kotlin Multiplatform**: 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 template used, the adopted project structure, and how to run tests in a multiplatform context;
- **Chapter 3 - Common Design and Implementation Strategy**: 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**: Describes the Retry mechanism functionality, its configuration, how it was implemented in both the library and as a plugin for the Ktor framework integration;

- **Chapter 5 - Circuit Breaker**: Same as the previous chapter, but for the Circuit Breaker mechanism;
- **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.

This chapter provides an overview of the technology, including its architecture, template structure, and other aspects relevant to the development of a multiplatform library.

2.1 Architecture

A KMP 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 KMP 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. The official Kotlin Multiplatform template [15] provides a project structure that includes the necessary configurations for building, testing, and deploying a multiplatform library for most platforms.

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 [16].

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

- **build.gradle.kts** - Defines a project's build configuration. In KMP projects, this file is used to define a project's targets, dependencies in the respective source sets, and additional configurations if needed;

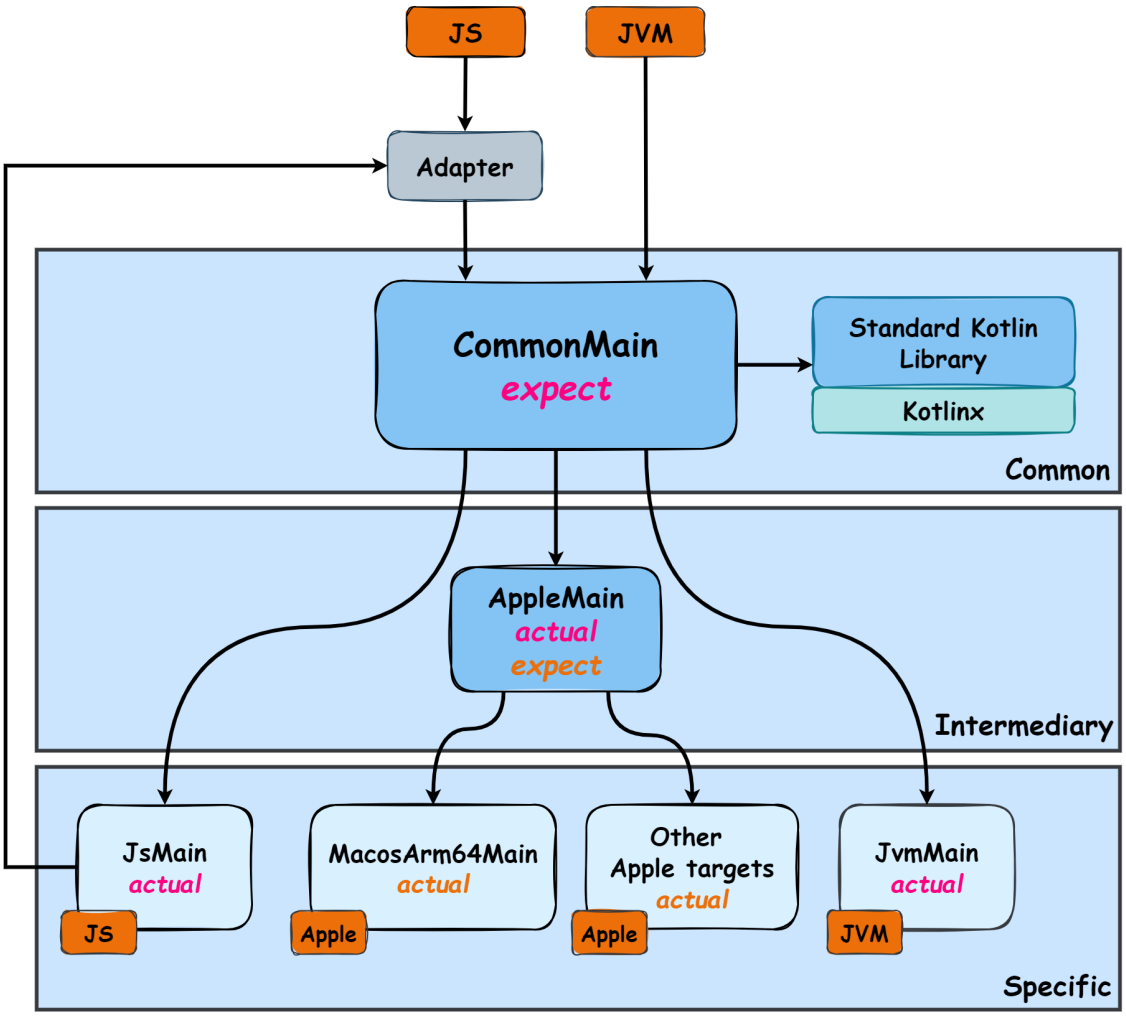

Figure 2.1: Example of a KMP project architecture.

- **settings.gradle.kts** - Defines a project's structure and modules it contains.

In Gradle, tasks are the smallest unit of work that can be executed and are used to perform specific actions. The templated uses the KMP plugin which includes several pre-configured Gradle tasks to facilitate building, testing, and managing a 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 [17] for continuous integration and continuous deployment (CI/CD) [18]. 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. Runs on push and pull request git events to the default branch;
- **deploy.yml** - deploys the library artifacts to a repository in Maven Central [19], following a pre-defined authentication configuration.

2.1.4 Folder Structure

The template 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;
- **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 a project, as dependencies, in a centralized manner (regularly known as version catalog [21]).
- **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 subprojects common 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 structure was adopted for developing a KMP library:

- **<kmp_package_name>**: name of the KMP 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* [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

This chapter outlines the shared design principles of the implemented mechanisms and the development approach used to integrate them with Ktor. A comprehensive review and analysis of existing mechanism implementations 1.3 were conducted to identify common design elements and relevant features that could be incorporated into the design of the new mechanisms.

3.1 Mechanism Model

The common design elements identified in the existing mechanisms were used to define a model that represents the mechanisms' core components. These components form the foundation of all future mechanism's implementation, 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 behaviour (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* 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, and provides a convenient way to wrap code blocks with the mechanism's behaviour;
- **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 behaviour and provide additional features in an HTTP context.

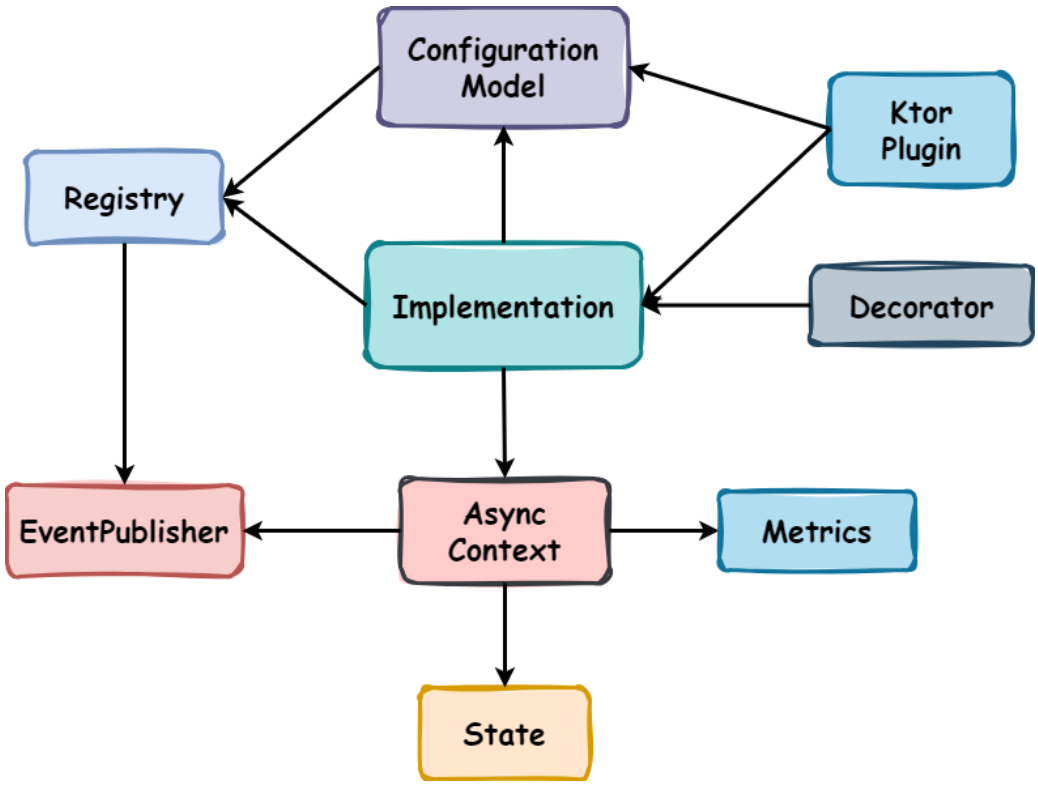

Figure 3.1: Mechanism Model

3.1.1 Configuration Design

Since the start of the project, the Configuration component has been designed to use the builder creational pattern [23, 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 be constructed 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 implemented using contination passing style (CPS) [26].

Thread

In computer science, a thread [27] 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. 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 (uses a state machine to manage its execution). 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 also published asynchronously. To achieve this, the Kotlin Coroutines' asynchronous primitive Flow [28] was used to emit such events.

Flow

A Flow in Kotlin is a type of asynchronous data stream that can emit multiple values sequentially. It is typically cold, meaning it starts emitting values only when collected, ensuring that each collector receives the full sequence of values from the start. In contrast, hot streams which emit values regardless of active collectors, allowing multiple consumers to observe ongoing data emissions [29, 28].

Flows are distinct from Sequences [30] in Kotlin. While both can handle multiple values over time, Flows are designed for asynchronous operations and do not block the calling thread when collecting values. Sequences, however, are synchronous and computed on demand (lazy), blocking the calling thread during value computation.

Implementation

In a mechanism, events 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.

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

- Receive all events emitted by the mechanism;
- 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. However, the cancellation of the registered listeners up to a given time does not affect later registrations.

3.2 Decoration

In mathematics, function composition is an operation that takes two functions f and g, and produces a function $h = g \odot f$ such that $h(x) = g(f(x))$.

In functional programming, a high-order function [31] is a function that takes one or more functions as arguments and/or returns a function as its result.

A decorator is a structural design pattern that attaches additional responsibilities to an object dynamically using aggregation/composition to provide a flexible alternative to subclassing for extending functionality [24].

In the context of these mechanisms, a decorator is a high-order function that wraps an operation (e.g., method, function) with the behaviour of the mechanism. The decorator needs to abide by the decoration principle, which states that a decorator should not modify the operation's signature (i.e., name, parameters and return value), only alter its behaviour. For callers of the operation, the decorator should be transparent (as if the operation was not decorated).

3.2.1 Operation Types

An operation represents a unit of work that can be decorated with the mechanism's behaviour (e.g., a function that makes an HTTP request).

In early development stages, the mechanisms were designed to support only one type of operation: a function that takes no arguments and returns a value. But this design was too restrictive, as it did not allow for operations that receive additional arguments. To address this limitation, the operation types were redefined to support different scenarios:

- **Supplier**: Accepts no arguments and produces a result. Based on Java's Supplier functional interface [32];
- **Function**: Accepts one argument and produces a result. Based on Java's Function functional interface [33];
- **BiFunction**: Accepts two arguments and produces a result. Based on Java's BiFunction functional interface [34].

Since the operations were going to be used in potential asynchronous contexts, the operation types were redefined to be suspendable (i.e., representing potentially suspendable operations).

3.2.2 Operation Context

In later stages of development, it was considered to give the aforementioned operation types context-awareness, meaning that the operations would have access to the mechanism's execution context in an immutable state (i.e., for read-only purposes).

As a result, and to maintain backwards compatibility with the existing operation types (which allowed for method references [35] to be decorated), a new family of operation types was created with a `Ctx` prefix (e.g., `CtxSupplier`). These new operation types receive the mechanism's execution context as an additional argument.

3.3 Ktor Integration

Integration with Ktor presented an opportunity to apply the mechanisms in a real-world scenario and validate their design and implementation. But before integrating the mechanisms with Ktor, it is necessary to understand Ktor's architecture and how to extend its functionality using plugins.

3.3.1 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.

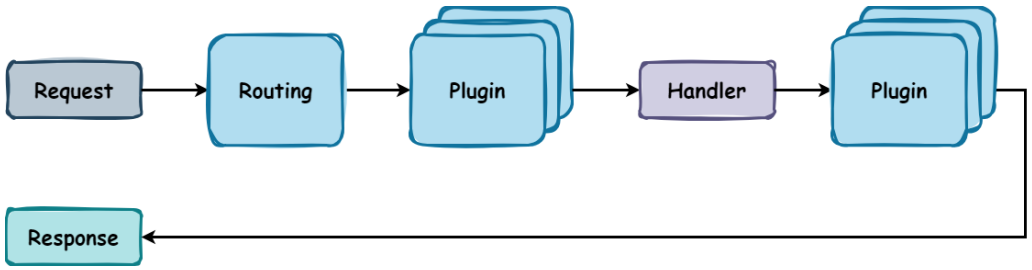

Figure 3.2: Ktor Server Architecture

On the server side, as a request comes in, it goes through a series of steps [36]:

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

3.3.2 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. 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 number, phases, and purpose. Tables 3.1 and 3.2 show the phases of the server and client pipelines, respectively.

3.3.3 Custom Plugins

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

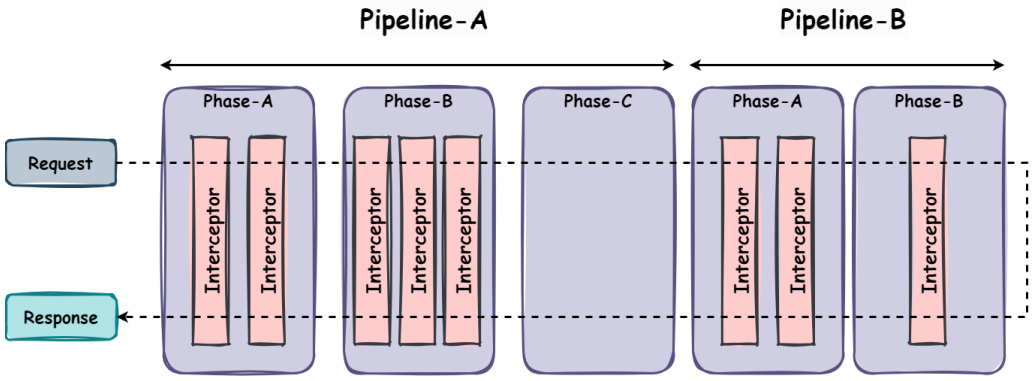

Figure 3.3: Ktor Pipeline Example

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

Since Ktor `2.0.0`, the custom plugin API has been simplified [37, 38] 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`, `onCallRespond`), which intercept the related phases of the pipeline.

3.4 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 were taken:

1. Implement the Mechanism Model, including all its components and any additional components required by the mechanism;
2. Write tests for the implemented mechanism;
3. Ensure that the mechanism works correctly for all targets;
4. Implement the Ktor Plugin that uses the mechanism;
5. Write tests for the Ktor Plugin; Due to time constraints, unit tests and integration tests were not conducted; however, functional tests [39] were performed using a real Ktor use case application;

Chapter 4

Retry

This chapter describes the retry mechanism, its functionality, configuration, how it was implemented in both the library and as a plugin for the Ktor framework integration.

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"* [40].

The retry mechanism, as illustrated in Figure 4.1, is particularly useful when the application is interacting with services that are prone to temporary (transient) 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) [40].

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.

When an application detects a failure (e.g., when sending a request to a remote service), it can handle it using the following strategies [40]:

- **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;

- **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 (wait duration) 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);

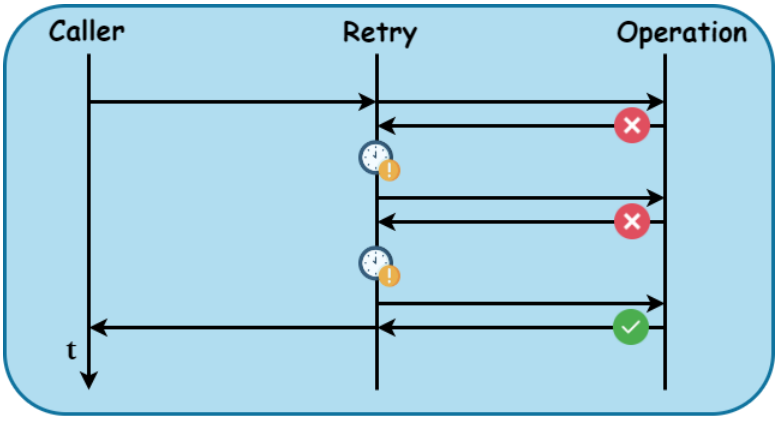

Figure 4.1: Retry Execution Example

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, attempts=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*. 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, attempts=4, result=[1, 2, 4, 8]`);
- **Custom Delay**: This strategy uses a custom function to calculate the delay duration based on the retry attempt number and the last exception caught (if any), allowing for more complex delay strategies (e.g., a delay that increases linearly for the first 3 attempts, then exponentially for the next attempts).

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, introducing a *jitter* can help mitigate the *thundering herd problem* by randomizing the calculated delay duration. This randomization prevents multiple clients from synchronizing their retries of the same operation, thereby spreading out the retries over a period of time.

4.2 Configuration

The retry mechanism can be configured using a dedicated configuration builder with the properties listed in Table 4.1.

Table 4.1: Configuration Properties for `RetryConfigBuilder`

Config Property	Default Value/Behaviour	Description
`maxAttempts`	`3`	The maximum number of attempts (including the initial call as the first attempt).
`retryPredicate`	`throwable -> true`	Predicate to determine if the operation should be retried based on the caught throwable.
`retryOnResultPredicate`	`result -> false`	Predicate to determine if the operation should be retried based on the result of the operation.
`delayStrategy`	`Exponential[ initialDelay=500ms, multiplier=2.0, maxDelay=1m]`	The strategy for calculating delay between retries.
`resultMapper`	`Rethrow throwable if any or return result unmodified`	Function to map the result or exception after the retry mechanism cannot be used any more to retry an operation

The default configuration can be overriden by calling the respective builder methods. The values that compose the default configuration are the most common and recommended for most use cases, but they can be adjusted to better suit the application's needs.

4.2.1 Custom Delay Provider

A custom delay provider can be used to implement a custom delay strategy that differs from the standard delay strategies (see Section 4.1.1).

A delay strategy determines the next wait time between retry attempts, while a delay provider executes the actual waiting period by pausing or blocking (dependends on the implementation) the process for the specified duration.

For example, for testing purposes, the custom delay provider might be implemented to not delay using real time, but instead, immediately continue the execution, and only keeping track of the time that would have been spent waiting (virtual time). This can be useful for speeding up tests that involve retry mechanisms, as it allows for faster execution without actually waiting for the delay duration.

A custom delay provider also allows maintaining external state (e.g., additional configuration through other dependencies) that can be used to calculate the delay duration, which can't be achieved with the standard delay strategies.

4.3 Implementation Aspects

TODO: Describe what deviates from the Mechanism model

4.3.1 Ignoring Exceptions

The idea of ignoring exceptions that shouldn't trigger a retry was initially considered, by adding an extra configuration property (as offered by Resilience4j [41]). However, it was decided that the retry predicate should be the only policy for determining if an operation should be retried based on the caught exception.

4.3.2 Result Mapper

During the initial development phases, if an operation failed after multiple retries, the last caught exception was always rethrown, and the original result was returned without any modifications. This approach is sufficient for most use cases, and most studied implementations of the retry mechanism follow this pattern, but it was decided to provide a more flexible solution (e.g., the caller doesn't want a specific exception to be thrown, but instead wants to log it only). To address this, an exception handler was introduced to the configuration. This handler allowed for additional processing of the caught exception (e.g., decide if it should be rethrown or logged), but still did not provide a way to modify the result of the underlying operation.

To provide even more flexibility, a result mapper was subsequently added to the configuration. This mapper enables customized handling of the result or exception once the retry mechanism could not be used any more, allowing for more complex scenarios to be managed while keeping the already established functionality intact. Additionally, the result mapper can be reconfigured per decoration.

4.3.3 Retry Context

The retry context represents the asynchronous context mechanism's model component implementation (see Section 3.1.3), which is responsible for managing the state and controlling the flow of retryable operations. It maintains internal state information, such as the number of retry attempts and the last-caught exception, and has access to the configuration settings defined for the retry mechanism. Additionally, it emits events to potential listeners.

In the later stages of development, the retry context was made available to the operation being retried (see Section 3.2.2), allowing it to access the context and potentially alter its behaviour based on its state.

Event Emission

The retry context emits the following events, when the operation:

- has succeeded and the retry mechanism is no longer needed;
- is about to be retried;
- has failed, and the retry mechanism cannot be used any more. Two different event types can be emitted in this case based on if the last-caught exception was expected or not.

4.3.4 Execution Flow

A retry mechanism can be used to decorate an operation. When the operation is executed, a new retry context is created (per-method invocation). The outcome of the operation—success or failure—is determined by the policies defined in this configuration.

When the underlying operation is executed, the retry context captures any exception or result that occurs. If the operation fails, the retry context consults the policies defined in the configuration to decide if a retry should be attempted. If so, the operation is repeated after a delay, calculated based on the specified delay strategy. But if the retry context determines that the operation should not be retried, the operation is considered to have failed.

If the operation eventually succeeds through the retry mechanism, the retry is considered successful. In all cases where the retry mechanism cannot be used any more (e.g., the maximum number of attempts has been reached), the result mapper is called to map the final result or exception before it is returned to the caller.

This description can be visualized in the retry mechanism execution flow, as illustrated in Figure 4.2.

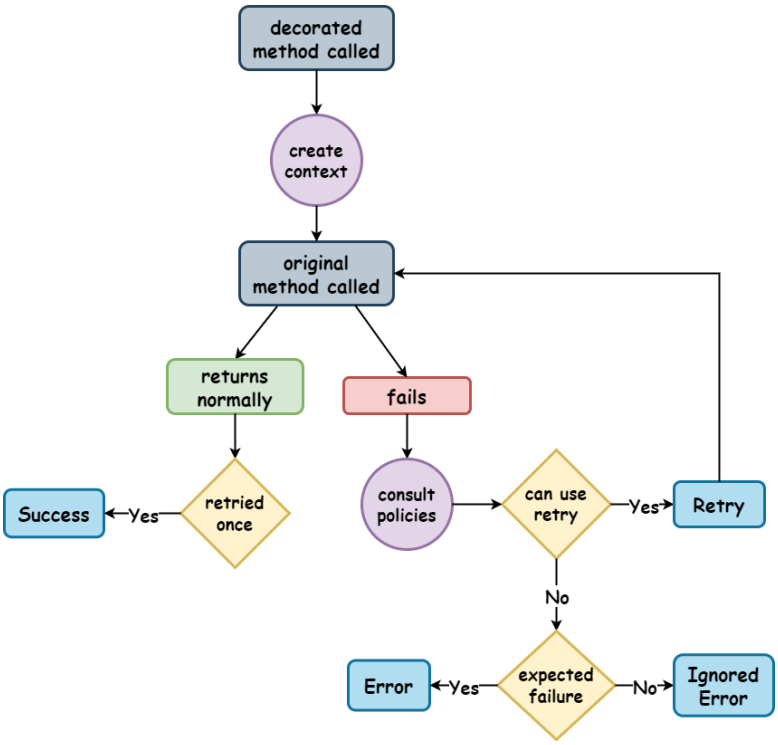

Figure 4.2: Retry Mechanism Execution Flow

4.4 Ktor Integration

The retry mechanism was integrated with Ktor by implementing a custom plugin that can be added to the application's Ktor pipeline, using the implemented retry mechanism.

4.4.1 Plugin Implementation

The plugin was designed for Ktor clients to enable automatic retries for HTTP requests. It intercepts the sending phase of a request, applying configurable retry logic based on conditions such as response status or exceptions.

Before retrying a request, the plugin copies the original request and propagates its completion state to the copy, ensuring completion synchronization between the original and the retry request (e.g., if the original request is cancelled, the retry request is also cancelled).

The plugin allows for two levels of configuration:

- **Global Configuration**: The configuration is set during the plugin installation and applies to all requests made by the client;
- **Per-Request Customization**: The configuration can be overridden for a specific request, allowing for more fine-grained control over the retry logic (e.g., disabling the retry mechanism for a specific request).

A listener to log all events emitted by the retry context is also enabled, which cancels its subscription when the request is completed.

4.4.2 Configuration

The retry Ktor plugin can be configured by providing a configuration builder with the properties listed in Table 4.2.

Table 4.2: Configuration Properties for `RetryPluginConfigBuilder`

Config Property	Default Value/Behaviour	Description
`maxAttempts`	`3`	The maximum number of attempts (including the initial call as the first attempt).
`retryOnExceptionPredicate`	`throwable -> true`	Predicate to determine if the operation should be retried based on the caught throwable.
`delayStrategy`	`Exponential[ initialDelay=500ms, multiplier=2.0, maxDelay=1m]`	The strategy for calculating delay between retries.
`retryOnCallPredicate`	`(request, response) -> retries if a 5xx response is received from a server`	Predicate to determine if an HTTP call should be retried based on the respective request and response.
`modifyRequestOnRetry`	`(requestBuilder, attempt) -> no-operation`	Callback to modify the request between retries (e.g., add a header with the current attempt number).

Additionally, methods relevant to an HTTP context were added to the configuration builder to simplify the configuration process:

- `retryOnServerErrors`: Retries the HTTP call if the response status code is in the range: 500-599;
- `retryOnServerErrorsIfIdempotent`: Similar to `retryOnServerErrors`, but only retries if the request method is idempotent [42] (e.g., `GET, PUT, DELETE`);
- `retryOnTimeout`: Retries the HTTP call if the exception thrown is a timeout exception (e.g., connection timeout exceeded).

Chapter 5

Circuit Breaker

This chapter describes the circuit breaker mechanism, its functionality, configuration, how it was implemented in both the library and as a plugin for the Ktor framework integration.

5.1 Introduction

The circuit breaker mechanism is a *"resilience pattern that prevents an application from performing an operation that is likely to fail. By allowing it to continue without waiting for the fault to be fixed or wasting CPU cycles while it determines that the fault is long-lasting "* [43].

When an application calls a remote service, there's always a risk of failure (e.g., network issues, service unavailability, or timeouts). Some of these faults are temporary (transient) and resolve quickly, but others can be long-lasting and severe (e.g., a complete loss of connectivity, complete failure of a service).

Moreover, if a service is under heavy-load, a failure in one part of the system might lead to cascading failures. For example, if an operation is set to time out and fail after a certain period, multiple concurrent requests could be blocked until the timeout expires. These blocked requests consume critical resources like memory, threads, and database connections, potentially leading to system-wide failures [43]. In such cases, it's more efficient for the application to quickly acknowledge the failure and handle it appropriately, rather than repeatedly retrying the operation.

5.1.1 Relation to Retry Mechanism

The circuit breaker mechanism is related to the retry mechanism, as both are used to handle faults that might occur when connecting to a remote service or resource. However, the two mechanisms are used in different situations.

The retry mechanism is used to retry an operation in the expectation that it'll succeed, while the circuit breaker mechanism is used to prevent an application from performing an operation that is likely to fail [43].

An application can combine these two mechanisms by using the retry mechanism to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any failures returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a failure is not transient [43].

5.1.2 State Machine

The circuit breaker mechanism works as an electrical circuit breaker, which is a safety device that automatically stops the flow of electric current in a circuit as a safety measure (e.g., to prevent a fire) when it detects a fault condition. In contrast to the electrical circuit breaker which needs to a manual reset, the circuit breaker mechanism automatically resets itself after a predefined period of time.

The default implementation of the circuit breaker uses a state machine with three states:

- **Closed** - The circuit allows the operation to execute and records its execution result (i.e., success or failure). If a configurable failure threshold is reached, the circuit transitions to the open state;
- **Open** - The circuit does not allow the operation to be executed. Instead, a predefined failure message is returned. The circuit remains open for a predefined period of time, after which it transitions to the half-open state. This timer is used to allow the system to recover from the fault condition;
- **Half-Open** - The circuit allows a limited number of requests to execute the operation. If these requests are successful, it's assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the `Closed` state (resetting the failure counter). If any request fails, the circuit breaker assumes that the fault is still present, so it reverts to the `Open` state (restarting the timeout timer).

The `Half-Open` state is useful to prevent a recovering service from suddenly being flooded with requests. As a service recovers, it might be able to support a limited volume of requests until the recovery is complete, but while recovery is in progress a flood of work can cause the service to time out or fail again [43].

Other implementations might have additional states (e.g., for maintenance, testing, metrics purposes), usually manually triggered, such as:

- **Forced Open** - The circuit is always open, preventing the operation from executing;
- **Forced Closed** - The circuit is always closed, allowing the operation to execute as if the circuit breaker were not present.

5.1.3 Operation Execution

Regarding the underlying operation execution, it's important to note that the circuit breaker does not synchronize the received calls (i.e., it does not prevent multiple threads from invoking the operation concurrently when the circuit is in a state that allows the operation to be executed).

To restrict the number of concurrent calls to the operation, a bulkhead [44] should be used in combination with the circuit breaker.

5.1.4 Recording Execution Results

The circuit breaker mechanism records the results of the operation executions to determine when to open the circuit. This recording uses a sliding window [45] which can be implemented in different ways:

- **Count-based** - The circuit breaker opens when the number of failures exceeds a predefined threshold (e.g., 50% of the last 10 operations failed, assuming a window size of 10).
- **Time-based** - The circuit breaker opens when the failure rate exceeds a predefined threshold over a specific period of time (e.g., 50% of the operations failed within the last 10 seconds, assuming a window size of 10 (1-second intervals)).

The time-based sliding window is used to track the outcomes of recent operation executions over a specific time interval, making decisions based on recent performance rather than solely relying on historical data (which might not be relevant in some cases).

As newer results are recorded, older results are removed from the window, keeping a FIFO (First In, First Out) ordering. This approach ensures that the window size remains constant and that the circuit breaker can make decisions based on the most recent data.

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] Android Developers. Kotlin-first android development. https://developer.android.com/kotlin/first?hl=en, 2024. [Online; accessed 7-March-2024].

[5] 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].

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

[7] JetBrains. Kotlin Multiplatform is Stable, 2024. [Online; accessed 28-May-2024].

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

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

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

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

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

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

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

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

[16] Gradle contributors. Gradle. https://gradle.org/, 2024. [Online; accessed 22-May-2024].

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

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

[19] Aaron Linskens. The history of maven central and sonatype: A journey from past to present, November 14 2023. [Online; accessed 28-May-2024].

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

[21] Gradle. Gradle user manual: Java platform plugin, 2024. [Online; accessed 28-May-2024].

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

[23] Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 3rd edition, 2017. [Online; accessed 23-May-2024].

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. [Online; accessed 28-May-2024].

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

[26] Saverio Perugini. *Programming Languages: Concepts and Implementation*. Jones Bartlett Learning, 2021. [Online; accessed 26-May-2024].

[27] Oracle. Thread. https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html, 2024. [Online; accessed 26-May-2024].

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

[29] Android. Stateflow and sharedflow. https://developer.android.com/kotlin/flow/stateflow-and-sharedflow?hl=en. [Online; accessed 28-May-2024].

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

[31] David Mertz. *Functional Programming in Python*. O'Reilly Media, Inc., 2015. [Online; accessed 26-May-2024].

[32] Oracle. Supplier. https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html, 2024. [Online; accessed 26-May-2024].

[33] Oracle. Function. https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html, 2024. [Online; accessed 26-May-2024].

[34] Oracle. Bifunction. https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html, 2024. [Online; accessed 26-May-2024].

[35] Oracle. Method references. https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html, 2024. [Online; accessed 26-May-2024].

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

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

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

[39] 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].

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

[41] Resilience4j contributors. Resilience4j: Retry. https://resilience4j.readme.io/docs/retry, 2024. [Online; accessed 26-May-2024].

[42] MDN Web Docs. Idempotent - MDN Web Docs, 2024. [Online; accessed 26-May-2024].

[43] Microsoft. Circuit Breaker pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker, 2023. [Online; accessed 30-May-2024].

[44] Microsoft. Bulkhead pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead, 2023. [Online; accessed 30-May-2024].

[45] GeeksforGeeks. Window sliding technique, 2024. [Online; accessed 30-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.