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 fault-tolerant.

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

- **Fault Tolerance**: A fault-tolerant service is a service that is able to maintain all or part of its functionality, or provide an alternative, when one or more of its associated components fail. The user does not observe see any fault except for some possible delay during which failover occurs.

- **Fault Resilience**: A fault-resilient service acknowledges faults but ensures that they do not impact committed data (i.e., the database may respond with an error to the attempt to commit a transaction, etc.).

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

1.2 Resilience Mechanisms

Over the years, several resilience mechanisms have been developed to help implemented build more robust and reliable systems. These mechanisms provide a set of tools and strategies to handle the inevitable occurrence of failures. Some of the most common mechanisms are described in table 1.1.

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

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

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

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

1.3 Technologies

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

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

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

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

1.4 Project Goal

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

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

1.5 Related Work

1.5.1 Ktor

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

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

1.5.2 Other Solutions

Traditional Libraries

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

Table 1.2: Examples of libraries that provide resilience mechanisms.

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

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

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

Arrow Library

The Arrow library, which presents itself as the functional companion to Kotlin's standard library, focuses on functional programming and includes, among other modules, a resilience library. This library implements three of the most critical design pattern around resilience [15]: retry and repeat computations using a *Schedule*, protect other services from being overloaded using a *CircuitBreaker*, and implement transactional behavior in distributed systems in the form of a *Saga*.

1.6 Document Structure

This document is structured as follows:

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

Chapter 2

Kotlin Multiplatform

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

2.1 Project Structure

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

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

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

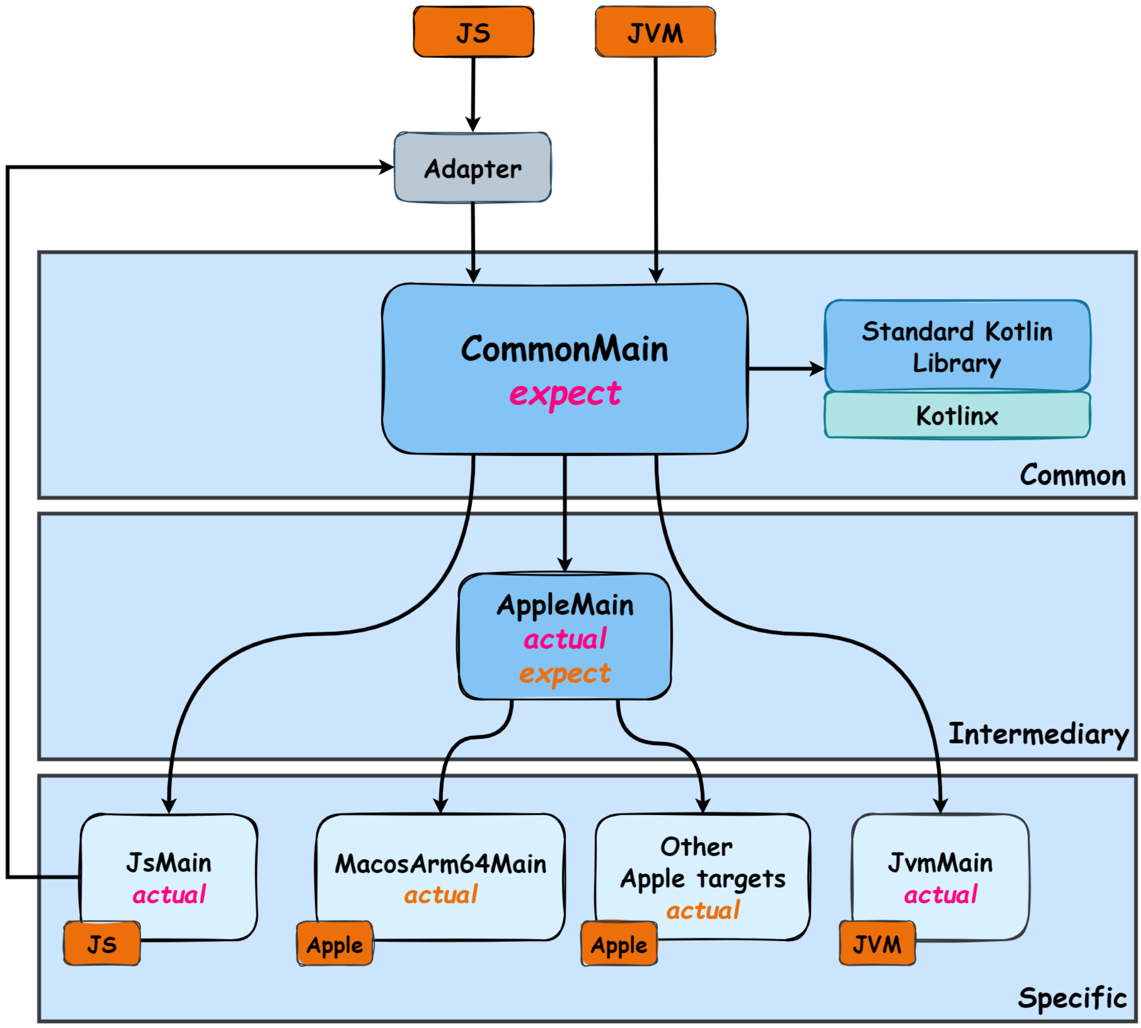
Mention kotlin multiplatform project template github repository Mention gradle project (divide in modules, gradle build file)

2.2 Platform-Dependent Code

Since the main goal of *KMP* is to share code across multiple platforms, the code should be written in a way that is as mutch platform-independent as possible (i.e., aggregating as much code as possible in the hierarchically superior 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*);

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



- The libraries available in the common category (i.e., *Standard Kotlin Library*, libraries from *Kotlinx*) do not cover the desired functionalities and reducing third-party dependencies is a priority;
- 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* or subset of platforms, the mechanism ***expect/actual*** [16] is used. This mechanism allows defining the code to be implemented in an abstracted way and its implementation, respectively.

2.3 Running Tests

2.4 Other Aspects

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

Chapter 3

Common Design and Implementation Strategy

For all mechanisms

3.1 Design Aspects

All the design and implementation aspects that are common to all mechanisms - use mechanism model

3.2 Implementation Aspects

- configuration - decoration - ktor pipeline plugin integration

Chapter 4

Retry

4.1 Introduction

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

4.2 Configuration

- mention default values and why they were chosen

4.3 Implementation Aspects

4.4 Ktor Integration

Chapter 5

Circuit Breaker

5.1 Introduction

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

5.2 Configuration

- mention default values and why they were chosen

5.3 Implementation Aspects

5.4 Ktor Integration

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A

Appendix Example

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.