JOHANNES KEPLER
UNIVERSITÄT LINZ

Submitted by
Gregor Lang

Submitted at
Institut für Systemsoftware

Supervisor
a.Univ.-Prof. Dipl.-Ing. Dr. Herbert Prähofer

06 2024

Ktor Reactive Server Applications

Bachelor Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Bachelor's Program
Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

This thesis explores Jetbrains's Ktor framework, showcasing its architecture and capabilities through the development of a prototype RESTful service and briefly comparing it to the Spring Boot framework. Ktor, designed as an asynchronous framework utilizing Kotlin's coroutines, is characterized through its streamlined setup, route handling, and external system integration. This exploration is paralleled with a similar implementation in Spring Boot to underline the key differences in setup complexity and coding overhead.

The thesis primarily aims to clarify the benefits of Ktor for developing flexible and light weighted apps that are perfect for microservices and settings that require high performance. A brief comparison with Spring Boot illustrates Ktor's unique methodology and places it in the larger backend framework ecosystem. In addition to adding to the conversation on contemporary web development technologies, the thesis seeks to offer useful insights on selecting Ktor for certain project requirements and developer preferences.

Zusammenfassung

Diese Arbeit untersucht das Ktor-Framework von Jetbrains, indem sie dessen Architektur und Fähigkeiten durch die Entwicklung eines Prototyps eines RESTful-Services vorführt und es kurz mit dem bereits etablierten Spring Boot-Framework vergleicht. Ktor, das als asynchrones Framework konzipiert ist und die Coroutinen von Kotlin nutzt, ist durch seine einfache Einrichtung, Routenverwaltung und die Integration externer Systeme gekennzeichnet. Diese Untersuchung wird mit einer ähnlichen Implementierung in Spring Boot parallel geführt, um die wesentlichen Unterschiede in der Einrichtungskomplexität und dem Programmieraufwand zu unterstreichen.

Die Arbeit zielt in erster Linie darauf ab, die Vorteile von Ktor für die Entwicklung von flexiblen und leichten Apps zu veranschaulichen. Solche Anwendungen sind perfekt für Mikroservices und Umgebungen, die eine hohe Leistung erfordern. Ein kurzer Vergleich mit Spring Boot veranschaulicht die einzigartige Methodik von Ktor und ordnet sie in das größere Ökosystem der Backend-Frameworks ein. Zusätzlich zur Diskussion über aktuelle Webentwicklungstechnologien strebt die Arbeit an, nützliche Einblicke in die Auswahl von Ktor für bestimmte Projektanforderungen und Entwicklerpräferenzen zu bieten.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Ort, Datum	Unterschrift

Contents

1 Introduction

In a world where Webapps are the most popular application format and HTML and JavaScript are the most widely used programming languages [1], effective backend solutions are required. With users expecting quick responses and seamless experiences, developers need tools that can keep up. This is where Ktor, a toolkit for the Kotlin programming language, comes into play. It helps developers build applications that can handle many tasks simultaneously, making everything run smoother and faster. This thesis aims to explore Ktor, showing why it's useful for more than just websites and how it stands out compared to other tools.

Ktor uses asynchronous programming, which means it can do things concurrently without waiting for each task to finish before starting the next one. This approach is key for building applications that are efficient and can serve many users at the same time.

This thesis tries to showcase what Ktor offers, how it works for different types of applications, and its pros and cons through testing and real examples. The goal is to understand Ktor's role in modern application development and how it might be the right choice for certain projects. This thesis will give you a clear picture of Ktor, focusing on its features, potential use cases and briefly comparing it to the widely used Spring Boot framework.

The structure of the thesis is as follows:

- Chapter 2: Explains the principles and architecture of Ktor
- Chapter 3: Showcases the inner workings of the test application 'KtorChat'
- Chapter 4: Compares the the Ktor framework to the well established Spring Boot framework, using different examples and criteria
- Chapter 5: Contains a conclusion

2 Ktor

Ktor is a versatile and lightweight web framework developed by JetBrains, designed for building asynchronous and compact web applications in Kotlin. Released as an open-source project, Ktor uses Kotlin's expressive syntax and coroutines to offer a simple, but powerful API for developers.

Ktor's asynchronous nature enables efficient handling of concurrent requests, making it suitable for high-performance applications. It provides built-in features for routing, content negotiation, authentication, and more, simplifying the development process. Additionally, Ktor seamlessly integrates with different environments and platforms, making it adaptable to various deployment scenarios.

Its modular architecture allows users to integrate only the components they need, promoting flexibility and minimizing overhead and bloat. It supports both client and server-side development, making it a comprehensive solution for end-to-end based web application development.

In this chapter a short introduction to the main Kotlin language features used for Ktor and the basic principles of Ktor is given.

2.1 Usage of Kotlin language features

Ktor utilizes several features of Kotlin to provide a compact API for building web applications. Kotlin's concise syntax reduces boilerplate code compared to Java, while its null safety features help prevent the notorious NullPointerExceptions. It supports functional programming concepts, has seamless interoperability with Java, and offers coroutines for simplified asynchronous programming. These and many more features collectively make Kotlin a powerful and efficient choice for JVM-based development.
The following paragraphs explain a few Kotlin language features, which appear a lot of times when using Ktor.

Trailing lambda

The trailing lambda pattern allows you to move a lambda expression outside of a function's parentheses if it's the last argument in the function call. This syntax enhances readability, but does not provide any functional differences.
Following is a short example, which showcases how this pattern works.

```kotlin
data class Person(var name: String, var age: Int)

fun processPersons(people: Set<Person>, action: (Person) -> Unit) {
    people.forEach { p ->
        action(p)
    }
}
```

In the first line a simple `data class` is defined which represents a single person by its name and age. Secondly a function is defined which takes a `Set` of `Person` as its first argument and a consumer (a lambda with return type `Unit`) as second (last) argument. This function simply iterates through the given `Set` and executes the given lambda on the current `Person`.
When calling `processPersons` the trailing lambda pattern comes into play.

```kotlin
fun main() {
    val alex = Person("Alex", 5)
    val otto = Person("Otto", 8)

    // lambda passed inside parenthesis
    processPersons(setOf(alex, otto), {p -> println(p)})

    // lambda passed outside parenthesis
    processPersons(setOf(alex, otto)) { p -> println(p) }
}
```

As seen in the above example, the lambda can be put outside (Line 9) of the paranthesis encapsulating the function parameters instead of inside (Line 6).
As a side note, in line 4 in the first code snippet containing the declaration of the `processPersons` function, the `forEach` method also makes use of the trailing lambda pattern. When the lambda is the functions only parameter, the opening and closing parenthesis can also be omitted. More on how lambda functions work can be read in the official documentation [2].

Lambda with receiver

The lambda with receiver pattern allows you to define a lambda expression, which has a receiving object of a specific type. Then, the lambda with receiver has to be called for an object of that type. The lambda with receiver then is executed in the context of the receiving object, i.e., it can be addressed by the `this` pointer. This pattern is particularly useful when you want to add to the functionality of a type without extending it or when you need to operate within the scope of that type. Inside the lambda, the receiver object can be accessed directly without explicit qualifiers, leading to more fluent and readable code.
Reusing the `data class Person` from the previous example, the following snippet showcases how the lambda with receiver pattern works.

```kotlin
fun copyPerson(person: Person, block: Person.() -> Unit) : Person {
    val newPerson = Person(person.name, person.age)
    newPerson.block()
    return newPerson
}
```

A function with a parameter of type `Person` and a lambda with receiver type `Person` as second parameter is defined. This function copies the given `Person` and then executes the given lambda onto the new `Person` instance. The copied instance, which may have been modified by the lambda, is then returned. In this example we can again make use of the trailing lambda pattern when calling the function.

```kotlin
fun main() {
    val alex = Person("alex", 5)

    val olderAlex = copyPerson(alex) {
        // this : Person
        this.age = 20
        // 'this' can be omitted here
        // age = 20
    }
}
```

The person instance the lambda is executed on can be accessed via `this`. In this case, `this` is the instance created by line 2 of the `copyPerson` function.

This pattern appears at every stage of the creation of an API, including route definition, plugin configuration, and request response.

Coroutines

Ktor heavily utilizes Kotlin coroutines "under the hood" for handling asynchronous operations. Coroutines allow developers to write asynchronous code in a sequential manner, which makes the code easier to understand and maintain. Ktor mainly uses coroutines for handling HTTP requests and responses asynchronously, making it efficient and non-blocking.
A coroutine can be started from a so-called `CoroutineScope`. An instance of this class can be accessed via a lambda with receiver provided by the method runBlocking. `CoroutineScope` provides the method `launch`, which launches a single coroutine asynchronously. The following snippet demonstrates on how to create coroutines.

```kotlin
fun main() {
    runBlocking {
        // this : CoroutineScope
        launch {
            // this : CoroutineScope
            delay(5000)
            println("Second")
        }

        launch {
            // this : CoroutineScope
            delay(2000)
            println("First")
        }
    }
    println("Third")
}
```

Function `runBlocking` blocks the underlying thread as long as it takes for all coroutines inside the block to finish.
Function `delay` is a method provided by `CoroutineScope`, which pauses the coroutine for the specified amount of milliseconds.
This example would print "First" then "Second" and lastly "Third" to the command line.
An in-depth documentation about coroutines can be found in [3].

2.2 Server

This chapter explains how a server can be created using Ktor.

The function `embeddedServer` is used to construct a server. As parameters, this procedure accepts an engine, a port and a lambda with receiver type `Application`, which is used to configure the server. The server may then be started and the underlying thread blocked while it waits for requests by calling `start` on the created instance.

```kotlin
fun main() {
    embeddedServer(Netty, port = 8080) {
        // this : Application
        ...
    }.start(wait = true)
}
```

This example server would serve no routes. To add functionality to the server the class `Application` provides various methods, most importantly `install`. `install` is used to add plugins to the server, which is going to be explained in 2.4. Another important method is `routing`. This method is a shortcut for `install(Routing)` and is used to configure routes. This method accepts a lambda with receiver type `Routing`.

```kotlin
fun main() {
    embeddedServer(Netty, port = 8080) {
        // this : Application
        routing {
            // this : Routing
            ...
        }
    }.start(wait = true)
}
```

`Routing` provides a function for each of the various HTTP methods:

- GET
- POST
- PUT
- PATCH
- DELETE
- HEAD
- OPTIONS

The function's name is the lowercase name of the HTTP method. It accepts the path of the request as a `String` and a lambda with receiver type `PipelineContext` as arguments.

```kotlin
fun main() {
    embeddedServer(Netty, port = 8080) {
        // this : Application
        routing {
            // this : Routing
            get("/") {
                // this : PipelineContext
                ...
            }
        }
    }.start(wait = true)
}
```

In this case a route at root level with HTTP method GET is defined.

When the route is created, the processing of incoming requests is configured. The purpose of the class `PipelineContext` is to enable access to the field `call`. Numerous pieces of information about the request are provided by this field, such as the IP address of the requesting client, a timestamp, query parameters, and many more.

To respond to requests, the field `call` also provides functions such as `respondText`.

```kotlin
fun main() {
    embeddedServer(Netty, port = 8080) {
        // this : Application
        routing {
            // this : Routing
            get("/") {
                // this : PipelineContext
                println(call.request.local.localAddress)
                call.respondText("Recorded your IP address!")
            }
        }
    }.start(wait = true)
}
```

In this scenario, the IP address of the requester would be printed to the server's command line and a simple `String` would be returned to the client.

There are many additional ways to respond to a client, such as

- `respondHtml` responds with a HTML page.
- `respondFile` responds with a file.
- `respond` responds with a serialized object.

2.3 Client

In the following part similar to the server, it is explained how to create a client using Ktor.

To create a client, we simply create an instance of the `HttpClient` class inside our main method. This results in us creating a bare-bones client, with which we would be able to send requests.

```kotlin
fun main() {
    val client = HttpClient()
}
```

The most basic way to send requests is via the `HttpClient::request` method, which accepts the target host url as `String` and a lambda with receiver type `HttpRequestBuilder` as arguments.

```kotlin
suspend fun main() {
    val client = HttpClient()

    val response: HttpResponse = client.request("localhost:8080") {
        // this : HttpRequestBuilder
        method = HttpMethod.Get
    }
}
```

The `HttpRequestBuilder` provides multiple useful fields for building the HTTP request, like `host` or `port`, but in this case only the field `method` is used, which is set to the HTTP method GET.

Additionally, the suspend keyword was added to the main function. This is because requests are executed asynchronously. Alternatively, the request call could be inside a runBlocking block as seen in the following example.

```kotlin
fun main() {
    val client = HttpClient()

    runBlocking {
        val response: HttpResponse = client.request("localhost:8080") {
            // this : HttpRequestBuilder
            method = HttpMethod.Get
        }
    }
}
```

Ktor has built-in functions that make it easier to call the various HTTP Methods. The following line shows how it works for the method GET:

```kotlin
val response: HttpResponse = client.get("localhost:8080")
```

There are also predefined methods for the following HTTP Methods:

- POST
- PUT
- PATCH
- DELETE
- HEAD
- OPTIONS

2.4 Configuration

A core part of Ktor's principles is configurability and extensibility. This is achieved via so-called plugins. Plugins are used to extend the server or client with additional functionality. These are added with the `install` method and the plugin constant as parameter. There are plenty of pre-defined plugins provided by the Ktor library, but custom plugins can also be created. The following paragraphs show how plugins are installed and how custom plugins are created for both client and server.
Even though client and server plugins might look very similar, they are very different on a technical level and cannot be used interchangeably.

Client

Installation of plugins for the client is done while creating the `HttpClient` instance. The `HttpClient` constructor accepts a lambda with receiver type `HttpClientConfig` as receiver. This class is mainly responsible for providing the method `install`. Additionally this class provides useful fields like `followRedirects`, which determines if request calls should follow redirects, or `expectSuccess`, which if set to `true`, would terminate the client if a request returns a non-success HTTP status code. Function `install` accepts a `Plugin` instance, which is provided as a constant by Ktor, and a lambda with receiver type depending on the plugin. Inside this lambda the plugin configuration class provides fields and methods, which can be used to configure the plugin.

```kotlin
val client = HttpClient {
    // this : HttpClientConfig
    install(HttpTimeout) {
        // this : HttpTimeout.HttpTimeoutCapabilityConfiguration
        requestTimeoutMillis = 2000
    }
}
```

In the example above, the `HttpTimeout` plugin is installed and provides an instance of its configuration class, `HttpTimeout.HttpTimeoutCapabilityConfiguration`. This class provides the field `requestTimeoutMillis`, which sets the maximum round trip duration of requests to the specified value in milliseconds.

There are many more predefined and very useful plugins such as:

- HttpRequestRetry: retry request automatically on configurable conditions
- DefaultRequest: set certain request parameters by default for the `HttpClient` instance
- ContentNegotiation: automatically set content-type header and (de)serialize data from and to the server
- HttpCookies: automatically set cookies
- Auth: configure different authentication algorithms for automatic authentication

More plugins can be found in [4].

If the predefined plugins are not sufficient, custom plugins can be created via `createClientPlugin`, which has to return the `Plugin` instance, which is used as parameter for the `install` method.
`createClientPlugin` accepts a `String`, representing the plugin name and a lambda with receiver type `ClientPluginBuilder` as parameter. This lambda provides different hooks, for executing functions at particular times. These hooks include:

- `onRequest`: when a request is sent
- `transformRequestBody`: transform the request body before the request is sent
- `onResponse`: when a response is received
- `transformResponseBody`: transform the response body after receiving it.
- `onClose`: when the `HttpClient` instance is closed

To make the plugin configurable, `createClientPlugin` also accepts a method reference to the constructor of a configuration class as second parameter. This configuration class is a regular POJO. Inside the lambda with receiver type `ClientPluginBuilder` the instance of the configuration class can be accessed via the field `pluginConfig`. The following code snippet shows how a configurable custom plugin can be created.
First a configuration class `TestPluginConfig` is defined. This class consists of a single `String` field `text`. Then the function `createClientPlugin` is called with the name of the plugin, the method reference to the constructor of the class `TestPluginConfig` and a lambda with receiver type `ClientPluginBuilder<TestPluginConfig>` as its arguments. Inside the lambda the `onResponse` function is called and is configured to print the content of the field `text` from the instance of the configuration class provided by `pluginConfig` to the command line whenever a response is received.

```
class TestPluginConfig {
    var text = ""
}

val TestPlugin = createClientPlugin("TestPlugin", ::TestPluginConfig) {
    // this : ClientPluginBuilder<TestPluginConfig>
    val responseText = pluginConfig.text
    onResponse {
```

```kotlin
        // this : OnResponseContext
        println(responseText)
    }
}
```

To install this plugin we can simply call the `install` function, like with any ordinary plugin.

```kotlin
val client = HttpClient {
    // this : HttpClientConfig
    install(TestPlugin) {
        // this : TestPluginConfig
        text = "Hello World"
    }
}
```

If the client now receives a response from the server, the `String` "Hello World" is printed to the command line.

Server

Installation of plugins for the server is done when starting the server via the `embeddedServer` function. Besides the server engine and the port the server should run on, the `embeddedServer` function accepts a lambda with receiver type `Application`. As mentioned earlier, `Application` provides the function `install`. This function works similar to the one from the client, accepting a `Plugin` instance and a lambda with receiver type depending on the plugin as argument.

```kotlin
fun main() {
    embeddedServer(Jetty, 8080) {
        // this : Application
        install(ContentNegotiation) {
            // this : ContentNegotiationConfig
            gson()
        }
    }.start(wait = true)
}
```

In this example, the `ContentNegotiation` plugin is installed and provides an instance of its configuration class `ContentNegotiationConfig`. This class provides the method `gson`, which tells the server that serialized content is going to be expected in the JSON format and that it will use Google's JSON parser implementation to deserialize the incoming data.

There exists many more predefined plugins:

- WebSockets: enables full-duplex communication over a single TCP connection
- Sessions: enable persisting data between HTTP requests
- RateLimit: limits the amount of requests a client can make
- RequestValidation: provides the ability to validate a body of incoming requests

More plugins for the server can be found in [5].

The server also provides the possibility of creating custom plugins. This works very similar to the way it does for the client except that it uses a different method (`createApplicationPlugin`) for creating the plugin and that it uses different hooks, which include:

- `onCall`

- `onCallReceive`
- `onCallRespond`

The following example shows how a configurable plugin is created for a server. Besides using `createApplicationPlugin` instead of `createClientPlugin`, the creation of a plugin for the server follows the same steps as creating a plugin for the client, which can be seen in the previous section.

```kotlin
class TestPluginConfig {
    var text = ""
}

val TestPlugin = createApplicationPlugin("TestPlugin", ::TestPluginConfig) {
    val responseText = pluginConfig.text
    onCall {
        println(responseText)
    }
}
```

To install this plugin, like with the client, the `install` function is used.

```kotlin
fun main() {
    embeddedServer(Jetty, 8080) {
        // this : Application
        install(TestPlugin) {
            // this : TestPluginConfig
            text = "Hello World"
        }
    }.start(wait = true)
}
```

Whenever the server would receive a request, "Hello World" would be printed to the command line.

2.5 Release Overview

The following section will summarize all important releases of Ktor and briefly mention significant additions and changes to the framework. All the following data derives from the official Ktor github page [6].

2.5.1 Pre-Release

The first public version was 0.9.0 and was released in October 2017, which featured plenty of experimental content and added a lot of core features. This pre-release phase lasted until the official release date for the first major version of November 2019.

2.5.2 Major Version 1

1.0 (November 2018)

The initial release of Ktor provided a foundation for building asynchronous web applications in Kotlin. It included support for routing, HTTP clients, and various features for building both server and client applications.

1.1 (December 2018)

This release focused on improving stability and performance. It introduced new features like content conversion and improved routing capabilities. Additionally, it included bug fixes and updates.

1.2 (May 2019)

The 1.2 release brought enhancements to the client and server API, improved support for Kotlin 1.3, and introduced features like automatic response conversion.

1.3 (January 2020)

Key features of this release included improved support for WebSockets, experimental support for GraphQL, and updates to the client and server components.

1.4 (August 2020)

Version 1.4 introduced structured concurrency, improved testing support, and better integration with kotlinx.serialization. It also included numerous bug fixes and enhancements.

1.5 (December 2020)

The 1.5 release brought a significant update to the ktor documentation, making it more accessible and comprehensive. It also introduced support for Kotlin 1.4 and included various bug fixes and improvements.

1.6 (May 2021)

This release focused on providing better support for Kotlin 1.5, including the new kotlinx.coroutines API. It also introduced a new plugin system, allowing for easier extensibility.

2.5.3 Major Version 2

2.0 (June 2022)

In this release significant changes were introduced, including updates to Kotlin 1.6.20, migration to a version catalog, and the addition of WebSockets support in the Darwin engine. The plugins API underwent a major overhaul, and new features like the Single Page Application plugin were introduced.

2.1 (August 2022)

YAML configuration format support was added, along with features such as the ability to override HSTS settings per host and pattern matching for CORS origin. The release focused on addressing bug fixes, improving functionality.

2.2 (December 2022)

Ktor 2.2 focused on bugfixes, addressing issues like connect timeout, URL parsing with underscores, and content-type-related errors.

2.3 (April 2023, ongoing)

This version introduced features such as support for multiple config files and regex patterns in routing. Improvements included Apache 5 upgrade for ktor-client. The release aimed at enhancing functionality and resolving reported bugs across modules.

3 Sample Application: KtorChat

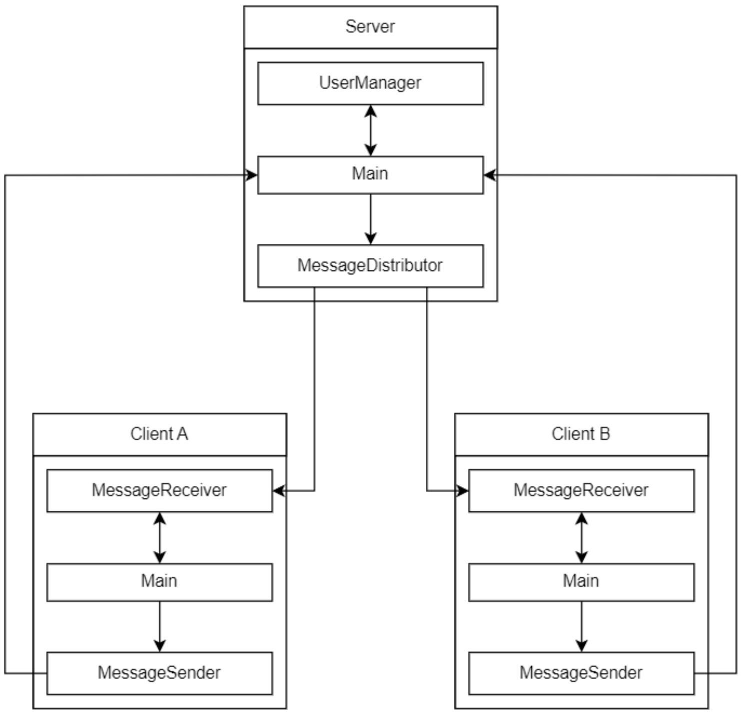
KtorChat is a CLI-based chatroom operating on a local area network. It is a prototype implementation, built for the purpose of showing features of Ktor. The implementation of KtorChat is far from ideal. Typically, chat systems leverage WebSockets and related technologies, to achieve better performance. This is a deliberate decision as choosing WebSockets would limit the ability to make use of a wide array of Http-based functionalities, making it impossible to demonstrate Ktor's feature set.
The application's design is as straightforward as possible. It just includes the features that are absolutely necessary for its operation. It serves as a model for understanding many elements of Ktor and their practical use. Furthermore, it is very adaptable, making it an ideal starting point for anyone curious in how Ktor features are utilized and implemented.

3.1 Structure

KtorChat consists of three modules.

1. **Common** contains various classes and definitions that are used by both the client and the server.
   - Definitions of data transfer objects
   - Custom plugins
   - Configuration constants
   - Content validation utility functions
2. **Server** transmits and receives messages to and from clients who are logged in.
3. **Client** receives messages from the server and publishes them on its interface, as well as sending messages entered by the user.

The diagram below depicts how each component of the three modules interacts with one another. These elements will be covered in further detail on the following pages.



The server's main component uses its `UserManager` instance to verify and store user information, and its `MessageDistributor` instance to get messages ready for delivery. The client's main component retrieves messages from its user interface and forwards them to the `MessageSender` instance to be sent to the server. Additionally, it publishes every message in the incoming message queue of the `MessageReceiver` instance to the user interface.

3.2 Project Setup

I choose to use Gradle as a dependency management tool for this project. Ultimately, it comes down to personal preference, although other tools like Maven could have also been used.
Gradle can be utilized with the Groovy DSL or the Gradle Kotlin DSL. To keep the project entirely in Kotlin, I prefer to use the latter. As a result, we use build files that end in `*.kts`.

3.2.1 build.gradle.kts

This file is where we add dependencies to our project. In the beginning the file should look something like the following:

```kotlin
val ktor_version: String by project
val kotlin_version: String by project

plugins {
    kotlin("jvm") version "1.8.21"
    id("io.ktor.plugin") version "2.3.1"
}

repositories {
    mavenCentral()
}

dependencies {
    // SERVER
    implementation("io.ktor:ktor-server-core-jvm:$ktorVersion")
    implementation("io.ktor:ktor-server-netty-jvm:$ktorVersion")

    // CLIENT
    implementation("io.ktor:ktor-client-core:$ktorVersion")
    implementation("io.ktor:ktor-client-cio:$ktorVersion")

    // add more here ...
}
```

The `gradle.properties` file contains the variables `ktor_version` and `kotlin_version`.
The versions of the JVM that Kotlin uses and the ktor plugins must be defined in the `plugins` block.
We specify which repositories Gradle should search for dependencies in the `repositories` block. We only use the central Maven repository in this project.
The actual dependencies required by the project are contained in the `dependencies` block. Dependencies must be added here in order to provide more functionality for the client and server.

3.3 Common

The Common module is divided into four components, each of which provides functionality required in both client and server.

3.3.1 Configuration

The `Configuration` class defines two constants, which specify on which ports the embedded servers run the client side and the server side.

```kotlin
class Configuration {
    companion object PORTS {
        const val CLIENT_PORT = 50001
        const val SERVER_PORT = 50000
    }
}
```

3.3.2 Data

The `Common` module defines four different data transfer objects.

```kotlin
data class MessageData(
    val targetUsers: Set<UUID>,
    val sourceUser: UUID,
    val message: String,
    var userName: String? = null
)
```

`MessageData` represents the contents and meta information of one message sent between server and client.

- `targetUsers` is a `Set` of `UUID` of users to whom the message is sent.
- `sourceUser` is the `UUID` of the sender of the message.
- `message` is the message text as a `String`.
- `userName` is the username of the user who sent the message as a nullable `String`.

```kotlin
data class UserData(val username: String)
```

`UserData` contains the information the client sends to the server when logging in for the first time. This object could be expanded by additional data such as e-mail address, birth date and many more.

```kotlin
data class LogoutData(val uuid: UUID)
```

`LogoutData` encapsulates the unique identifier of a user. This object is sent from the client to the server when logging out.

```kotlin
data class LoginResponse(val success: Boolean, val id: UUID?, val errorMessage: String?)
```

`LoginResponse` contains information about whether or not a login attempt was successful.
If it was successful, `success` will be `true`, `errorMessage` will be `null`, and `id` will contain the user's `UUID`.
If it failed, `success` will be `false`, `id` will be `null`, and `errorMessage` will contain a `String` with an error message detailing what went wrong.

3.3.3 Plugins

The `plugins` module defines a custom plugin called `ServerReadyCallback`.

Plugins use configuration classes which have no specific type and are used to pass settings from the installation process to the plugin itself. The following example depicts how such a class might look like.

```kotlin
class ServerReadyCallbackConfiguration() {
    var block: suspend CoroutineScope.(ApplicationEnvironment) -> Unit = {}

    fun callback(block: suspend CoroutineScope.(ApplicationEnvironment) -> Unit) {
        this.block = block
    }
}
```

The ServerReadyCallback plugin's configuration class contains an suspending receiver lambda for type CoroutineScope. This lambda is configured during the plugin installation process and is called once the server is ready to receive requests.

A plugin is created via the method `createApplicationPlugin()`, provided by the Ktor framework. This method accepts the plugin's name as a `String` and a lambda function that returns an instance of a configuration class.

In the following exaple the `ServerReadyCallback` plugin is created with its name and a method reference to the constructor of the configuration class as parameters.

```kotlin
val ServerReadyCallback = createApplicationPlugin(
    name = "ServerStatusLogging", createConfiguration = ::ServerReadyCallbackConfiguration
) {
    on(MonitoringEvent(ServerReady)) {
        runBlocking {
            pluginConfig.block(this, it)
        }
    }
}
```

The `ServerReadyCallback` plugin uses the `on()` handle to run the lambda contained in the instance of the configuration class.

There are multiple handles a plugin can use to access different stages of a call.

- `onCall` provides the ability to obtain request/response information, modify response parameters (for example, append custom headers), and so on.
- `onCallReceive` enables processing and transforming data received from the client.
- `onCallResponed` enables data transformation before it is sent to the client.
- `on()` allows the execution of specific hooks, such as a specific `MonitoringEvent`.

3.3.4 Validators

The `validators` module contains global methods which are used to configure a `RequestValidation` plugin.

The code snippet below shows an enum class with different levels of acceptable characters for a username. Each enum object has a predicate `test`, which is used to determine whether a `String` has illegal characters.

```kotlin
enum class AllowedUserNameCharacters(val description: String, val test: (String) -> Boolean) {
    ALL("all", { true }),
    ALPHANUMERIC("alphanumeric", { s -> s.all { it.isLetterOrDigit() } }),
    ALPHA("alphabetical", { s -> s.all { it.isLetter() } }),
}
```

The following method `validateUserData` validates requests that have a payload of type `UserData`.

```kotlin
fun RequestValidationConfig.validateUserData(
    maxUserNameLength: Int = 10,
    minUserNameLength: Int = 3,
    allowedUserNameCharacters: AllowedUserNameCharacters =
        AllowedUserNameCharacters.ALPHANUMERIC
) {
    this.validate<UserData> {
        val username = it.username

        if (username.length < minUserNameLength || username.length > maxUserNameLength) {
            return@validate ValidationResult.Invalid(
                "Username has to be between $minUserNameLength " +
                        "and $maxUserNameLength characters!")
        }

        if (!allowedUserNameCharacters.test(username)) {
            return@validate ValidationResult.Invalid(
                "Username can only contain " +
                        "${allowedUserNameCharacters.description} characters!")
        }

        return@validate ValidationResult.Valid
    }
}
```

It determines whether the username is between 3 and 10 characters long and tests whether the characters are letters or numbers via the .

The second method `validateMessageData` validates requests that have a payload of type `MessageData`.

```kotlin
fun RequestValidationConfig.validateMessageData(maxMessageLength: Int = 256) {
    this.validate<MessageData> {
        val message = it.message
        if (message.length > maxMessageLength) {
            ValidationResult.Invalid(
                "Message cannot be longer than $maxMessageLength characters!")
        }

        ValidationResult.Valid
    }
}
```

It determines whether the message text length exceeds the limit of 256 characters.

3.4 Server

The server consists of three parts.

- Main. Contains the entry point for the program
- MessageDistributor. Distributes incoming message to the the connected clients
- UserManager. Manages permissions, and login and logout of clients

3.4.1 Main

The main component is divided into the main method and the `MessageServer` class. The main method is responsible for launching the server and polling the `MessageServer`'s queue to forward the messages to the `MessageDistributor`. The MessageServer class processes three different requests.

- `login` and `logout`
  The login or logout payloads are reconstructed into objects the `UserManager` can handle. The server then waits for a response from the `UserManager` before returning an OK or Unauthorized Response to the client.
- `message`
  First, the server determines if the user specified as the source matches the IP address from which the request originated. If it does, the message is placed in a channel that is polled by a separate coroutine, and the `MessageDistributor`'s lone method distribute is called for each item.

Implementation (main)

The following code snippet depicts the implementation of the main method.

```kotlin
fun main() {
    val userManager = UserManager()
    val messageServer = MessageServer(userManager)
    val messageDistributor = MessageDistributor(
        onError = { _, uuid -> userManager.logout(uuid) },
        onTimeout = { _, uuid -> userManager.logout(uuid) },
        fallbackTargetProducer = { userManager.getAllUsers().minus(it.sourceUser) }
    )

    runBlocking {
        launch {
            println("Start MessageServer on Thread ${Thread.currentThread().name}")
            messageServer.start()
        }

        launch {
            println("Start DistributeMessage Loop on Thread ${Thread.currentThread().name}")
            for (msg in messageServer.messageChannel) {
                messageDistributor.distribute(msg) {
                    userManager.getUser(it)?.host
                }
            }
        }
    }
}
```

After creating `UserManager`, `MessageServer` and `MessageDistributor` instances, two seperate coroutines are launched. The arguments provided to the constructors of those classes are going to be explained in their respective chapters. The first coroutine starts the server. The second polls the `MessageServer`'s channel in an endless loop and forwards the messages to the `MessageDistributor`.

Implementation (MessageServer)

The `MessageServer` class is responsible for handling `login`, `logout` and `message` requests.

```
 1 class MessageServer(private val userManager: UserManager) {
 2     private val rateLimitLogin = "login"
 3     private val rateLimitMessage = "message"
 4     private val serverName = "SERVER"
 5     private val serverUuid = UUID.nameUUIDFromBytes(serverName.toByteArray())
 6
 7     val messageChannel = Channel<MessageData>()
 8
 9     fun start() =
10         embeddedServer(
11             Jetty,
12             port = Configuration.SERVER_PORT,
13             module = { module() }).start(wait = true)
14
15     private fun Application.module() {
16         installPlugins()
17         configureRoutes()
18     }
19
20     private fun Application.installPlugins() {
21         // ...
22     }
23
24     private fun Application.configureRoutes() {
25         // ...
26     }
27
28     private suspend fun sendServerMessage(message: String) {
29         // ...
30     }
31
32     private suspend fun ApplicationCall.respondIf(
33         payload: Any? = null,
34         statusCodeFail: HttpStatusCode = HttpStatusCode.Unauthorized,
35         check: () -> Boolean,
36         block: suspend () -> Unit
37     ) {
38         // ...
39     }
40
41     private suspend fun ApplicationCall.respondIfAuthorized(
42         userUuid: UUID, block: suspend () -> Unit) {
43         // ...
44     }
45 }
```

The `MessageServer`'s constructor receives an `UserManager` instance, which it uses to process and store user information.

In line 2 to 6 various `String` constants are defined, which are going to be used later on.

Line 7 contains the creation of the channel containing all processed messages.

The method `start()` creates and starts an embedded server. This embedded server instance is configured via the `module` method, which in turn calls `installPlugins` and `configureRoutes`. All of those methods are extension functions for the class `Application`, which provides important methods, like `install` and `routing`.

installPlugins

The `installPlugins` method is an extension function for the `Application` class that offers `install`, which is used to add plugins to the server. The plugin that should be added is specified by a constant, provided by the ktor framework, and passed to the `install` method as parameter.

```kotlin
private fun Application.installPlugins() {
    install(ContentNegotiation) {
        gson {
            serializeNulls()
        }
    }
    install(RateLimit) {
        register(RateLimitName(RATE_LIMIT_LOGIN)) {
            rateLimiter(limit = 5, refillPeriod = 10.seconds)
        }
        register(RateLimitName(RATE_LIMIT_MESSAGE)) {
            rateLimiter(limit = 20, refillPeriod = 10.seconds)
        }
    }
    install(RequestValidation) {
        validateUserData()
        validateMessageData()
    }
    install(StatusPages) {
        exception<RequestValidationException> { call, cause ->
            call.respond(HttpStatusCode.BadRequest, cause.reasons.joinToString())
        }
    }
    install(ServerReadyCallback) {
        callback {
            println("Server is running.")
        }
    }
}
```

`installPlugins` adds five different plugins to the server.

- `ContentNegotiation` serves two primary purposes:
    1. Negotiating media types between the client and server using the `Accept` and `Content-Type` headers.
    2. Serializing/deserializing the content in a specific format. Ktor supports the following formats out-of-the-box: `JSON`, `XML`, `CBOR`, and `ProtoBuf`.

  The `gson()` method in the plugin's configuration block instructs the server to only accept content in the format of `JSON` and to serialize/deserialize it using Google's Gson library.
- `RateLimit` establishes various limitations that restrict the amount of traffic that the server can process in a specific time frame. In this situation, routes that use the `RATE_LIMIT_LOGIN` limit can only be serviced at most five times per ten seconds.
- `RequestValidation` validates all incoming requests. The methods `validateUserData` and `validateMessageData` are both globally defined in the `common` package and are further explained in section 3.3.4.

- StatusPages is used to define custom status pages if exceptions occur while the server processes the request. When the RequestValidation plugin determines that a request is invalid, it throws a RequestValidationException. This would result in an HTTP code 500 response, indicating an internal server issue. This is avoided by enabling the StatusPages plugin to return HTTP code 400 (Bad Request) in the event of a RequestValidationException.
- ServerReadyCallback is a custom plugin that allows to specify a callback that is executed when the server is ready to receive requests. In this example a simple String is printed to the command line.

respondIf

This method is used to respond to calls when fulfilling a condition. The procedure of responding to calls will be discussed on the following pages when routes are defined.

```kotlin
private suspend fun ApplicationCall.respondIf(
        payload: Any? = null,
        statusCodeFail: HttpStatusCode = HttpStatusCode.Unauthorized,
        check: () -> Boolean,
        block: suspend () -> Unit
    ) {
        var code = statusCodeFail
        if (check()) {
            block()
            code = HttpStatusCode.OK
        }

        if (payload == null) {
            this.respond(code)
        } else {
            this.respond(code, payload)
        }
    }
```

respondIf takes four arguments:

1. payload is the nullable object that is sent back to the client as body of the response.
2. statusCodeFail represents the HTTP status code, which should be returned in case the predicate evaluates to false. It defaults to HTTP code 401 (Unauthorized).
3. check is the predicate that is going to be evaluated. If it evaluates to true, HTTP status code 200 (OK) is returned.
4. block is a suspending lambda, which is executed when the predicate evaluates to true.

respondIfAuthorized

This method utilizes respondIf to only respond HTTP status code 200 (OK) if the given UUID is authorized by the UserManager.

```kotlin
private suspend fun ApplicationCall.respondIfAuthorized(
    userUuid: UUID, block: suspend () -> Unit) {
    respondIf(check = { userManager.isAuthorized(userUuid, this.request.local.remoteHost) }) {
        block()
    }
}
```

sendServerMessage

This method puts a new MessageData object into the server's message channel.

```kotlin
private suspend fun sendServerMessage(message: String) {
    messageChannel.send(MessageData(emptySet(), SERVER_UUID, message, SERVER_NAME))
}
```

- emptySet() signals that this message should be sent to every connected client.
- SERVER_UUID represents the server's own UUID.
- message is the message to be broadcasted.
- SERVER_NAME represents the server's own username that is going to show in the chat room.

configureRoutes

configureRoutes, like installPlugins, is an extension function for the class Application, which also offers the method routing. routing accepts an extension lambda for the Routing class as an argument.

The class Routing offers methods such as get and post for defining routes that the server may serve. These methods take two arguments: a String describing the route and an extension lambda on the PipelineContext class. PipelineContext inhabits the field call of type ApplicationCall, which contains useful meta information about the client as well as methods for responding to requests.

```kotlin
private fun Application.configureRoutes() {
    routing {
        rateLimit(RateLimitName(RATE_LIMIT_LOGIN)) {
            post("/login") {
                val userdata = call.receive<UserData>()
                val response = userManager.login(userdata, call.request.local.remoteHost)
                call.respondIf(payload = response, check = { response.success }) {
                    sendServerMessage("${userdata.username} joined.")
                }
            }
        }

        post("/logout") {
            val data = call.receive<LogoutData>()
            call.respondIfAuthorized(data.uuid) {
                val username = userManager.getUser(data.uuid)?.username ?: "<???>"
                sendServerMessage("$username left.")
                userManager.logout(data.uuid)
            }
        }

        rateLimit(RateLimitName(RATE_LIMIT_MESSAGE)) {
            post("/message") {
                val message = call.receive<MessageData>()
                call.respondIfAuthorized(message.sourceUser) {
                    message.userName = userManager.getUser(message.sourceUser)?.username
                    messageChannel.send(message)
                }
            }
        }
    }
}
```

The server has three routes defined:

- `login`
  A client can connect to the server by initiating a `login` request. It is wrapped in a lambda passed to the `rateLimit` function, which implements the rate limit defined in the `installPlugins` method when installing the `RateLimit` plugin. When a `login` request reaches the server, the payload is deserialized into a `UserData` object. This object, together with the client's IP address, is used to log the client into the `UserManager`. The server then answers with the method `respondIf`, which sends a message to all clients via `sendServerMessage`, if the check succeeds.
- `logout`
  To disconnect from the server, the client sends a `logout` request, which includes a serialized `LogoutData` object as payload. The server then deserializes the object and responds to the client using the `respondIfAuthorized` method, which takes the client's `UUID` as an input. If the client was authorized, a message is sent to all clients via `sendServerMessage` and the client is logged out.
- `message`
  The client sends messages to the chat room by submitting a `message` request that includes a serialized `MessageData` object as payload. The server answers to the request using the `respondIfAuthorized` method. If the client was approved, the `MessageData` object is updated with the client's username, which was resolved from its `UUID` by the `UserManager`. Finally, the `MessageData` object is added to the server's message channel.

3.4.2 UserManager

The UserManager is in charge of all users and hosts that the server is currently serving. It is essentially a simple list in which users are added when they log in and removed when they log out. Metadata such as hostname, username, login time, and so on are also stored in the UserManager.

Implementation (User)

The `UserManager` is divided into two parts, the first being the User class. It is a simple class which holds information, like the name of the user, the IP address of the users host and the last time they logged in.

```kotlin
data class User(val username: String, val host: String, val lastLogin: Date = Date())
```

Implementation (UserManager)

The second part consists of the `UserManager` class itself. The following code snippet shows the implementation of the class.

```kotlin
class UserManager {
    private val users = ConcurrentHashMap<UUID, User>()
    private val usernames = ConcurrentSet<String>()
    private val hosts = ConcurrentSet<String>()

    fun login(userdata: UserData, host: String): LoginResponse {
        // ...
    }

    fun logout(uuid: UUID) {
        // ...
    }

    fun isAuthorized(userId: UUID, host: String): Boolean {
        // ...
    }

    fun getAllUsers(): Set<UUID> = users.keys

    fun getUser(userId: UUID): User? = users[userId]
}
```

The class is instantiated via the implicitly defined default constructor. Three different collections that allow concurrent access are used to store the information of all users.

- `users` is a map that ties a unique identifier (UUID) to an instance of the `User` class. This unique identity is utilized throughout KtorChat and acts as the primary identifying property.
- `usernames` and `hosts` function as caches, including all previously chosen usernames as well as the IP addresses of all clients that connected to the server. This manner, when a new user joins, it is faster to check to see whether a username is duplicated or if a host wishes to connect with a second client, which is prohibited in this application.

The methods `getAllUsers` and `getUser` are basic helper functions that obtain `UUID`s of all users as a set and a single User instance by their `UUID` from the users map, respectively.

login

`login` is called by the server when a new user joins the chat room. It returns a `LoginResponse` with information about whether the user was approved and, if not, the reason for the refusal.

```
1  fun login(userdata: UserData, host: String): LoginResponse {
2      val username = userdata.username
3
4      if (usernames.map { it.lowercase() }.contains(username.lowercase())) {
5          val loginResponse = LoginResponse(false, null, "Duplicate username")
6          println("Login failed: $username@$host (${loginResponse.errorMessage})")
7          return loginResponse
8      }
9
10     if (hosts.contains(host)) {
11         val loginResponse = LoginResponse(false, null, "Duplicate host")
12         println("Login failed: $username@$host (${loginResponse.errorMessage})")
13         return loginResponse
14     }
15
16     hosts.add(host)
17     usernames.add(username)
18
19     val uuid = UUID.nameUUIDFromBytes(username.encodeToByteArray())
20     users[uuid] = User(username, host)
21
22     println("Login success: $username@$host ($uuid)")
23     return LoginResponse(true, uuid, null)
24 }
```

The first if block from line four to eight checks if a user with the requested user was already logged in. The second if block from line ten to 14 check if a client is trying to log in a second chat client. If both checks fail, the client's username and host are added to the caches, and a `UUID` based on the username is generated and used as the key for the new User instance.

logout

`logout` is called by the server when a user leaves the chat room. The user is simply removed from both caches and the `users` map.

```kotlin
fun logout(uuid: UUID) {
    val user = getUser(uuid) ?: return

    usernames.remove(user.username)
    hosts.remove(user.host)
    users.remove(uuid)

    println("Logout success: ${user.username}@${user.host} ($uuid)")
}
```

isAuthorized

`isAuthorized` checks whether a user with the specified `UUID` is authorized to send requests behalf of the client associated with the IP address contained in `host`.

```kotlin
fun isAuthorized(userId: UUID, host: String): Boolean {
    val user = users[userId] ?: return false
    return user.host == host
}
```

First the method checks if the user is logged in by checking the `users` map. If it is, the IP address stored inside the `User` instance is compared to the specified one.

3.4.3 MessageDistributor

The `MessageDistributor` has only a single purpose which is to distribute a single message to a given list of target hosts. The message is then sent to each client's `MessageReceiver` instance.

Implementation

The code sample below demonstrates the class's implementation.

```kotlin
class MessageDistributor(
    val onError: (MessageData, UUID) -> Unit = { _, _ -> },
    val onTimeout: (MessageData, UUID) -> Unit = { _, _ -> },
    val fallbackTargetProducer: (MessageData) -> Set<UUID> = { emptySet() }
) {

    private var client: HttpClient = HttpClient {
        // ...
    }

    suspend fun distribute(message: MessageData, uuidToHostResolver: (UUID) -> String?) {
        // ...
    }
}
```

The constructor of the class has three optional arguments:

1. `onError` is a lambda which receives a `MessageData` object and a `UUID` as input. This lambda is executed when an error occurs when distributing a message.

2. `onTimeout` is lambda which also receives a `MessageData` object and a `UUID` as input. It is executed when a request of the `HttpClient` instance of the `MessageDistributor` runs into timeout.
3. `fallbackTargetProducer` is a lambda which receives a `MessageData` object as input. It is executed when the `targetUsers` field within the `MessageData` object that will be distributed is empty.

client

This `HttpClient` instance sends requests to each target `MessageReceiver`, using a serialized `MessageData` object as the body.

```kotlin
private var client: HttpClient = HttpClient {
    defaultRequest {
        contentType(ContentType.Application.Json)
    }
    install(ContentNegotiation) {
        gson {
            serializeNulls()
        }
    }
    install(HttpTimeout) {
        requestTimeoutMillis = 2000
    }
    install(HttpRequestRetry) {
        maxRetries = 2
        delayMillis { 5000 }
        retryOnExceptionIf { _, cause ->
            cause.unwrapCancellationException() is HttpRequestTimeoutException
        }
    }
}
```

Four different plugins are required by the `HttpClient`.

1. `DefaultRequest` is a plugin that sets the request's default parameters. The value of the HTTP header `Content-Type` is always set to `application/json`. This plugin might alternatively be installed using `install(DefaultRequest)`, however Ktor provides a shortcut with the method `defaultRequest`.
2. `HttpTimeout` throws an `HttpRequestTimeoutException` for any request whose roundtrip time from sending the request to receiving the response exceeds the specified timeout. In this case all requests that take longer than two seconds are discarded and an exception is thrown instead.
3. `HttpRetry` retries unsuccessful requests in response to a specified condition. In this case the request is retried every five seconds and at most two times when the call throws an `HttpRequestTimeoutException`.

distribute

As a short reminder from section 3.4.1, the `MessageDistributor` instance from the server's main method was instantiated the following way.

```kotlin
// ...
val messageDistributor = MessageDistributor(
    onError = { _, uuid -> userManager.logout(uuid) },
    onTimeout = { _, uuid -> userManager.logout(uuid) },
    fallbackTargetProducer = { userManager.getAllUsers().minus(it.sourceUser) }
)
// ...
```

The following function is responsible for distributing message to all clients which are connected to the server. Its two arguments are `message` and `uuidToHostResolver`.

- `message` is a `MessageData` object, and is going to be the body of the request.
- `uuidToHostResolver` is a lambda responsible for resolving a `UUID` belonging to a user to an IP address.

```
suspend fun distribute(message: MessageData, uuidToHostResolver: (UUID) -> String?) {
    val targets = message.targetUsers.ifEmpty { fallbackTargetProducer(message) }

    targets.forEach { uuid ->
        val clientReceiverHost = uuidToHostResolver(uuid) ?: return@forEach
        val response: HttpResponse = try {
            client.post("receive") {
                host = clientReceiverHost
                port = Configuration.CLIENT_PORT
                setBody(message)
            }
        } catch (e: HttpRequestTimeoutException) {
            println("Timout to $clientReceiverHost")
            onTimeout(message, uuid)
            return@forEach
        }

        if (response.status.value.let { it in 500..599 }) {
            println("Error to $clientReceiverHost")
            onError(message, uuid)
            return@forEach
        }

        println("Distributed message from " +
                "${uuidToHostResolver(message.sourceUser)} " +
                "to $clientReceiverHost! -> $response")
    }
}
```

The distribute function operates as follows:

1. If the set targetUsers from the MessageData object is empty, the function `fallbackTargetProducer` is called. This creates a set of all users that are signed in to the server, excluding the one sending the message.
2. Each target UUID is resolved to its IP address and used as the target host for the post request to the client's `MessageReceiver`.
3. If an `HttpRequestTimeoutException` occurs, the request is terminated and `onTimeout` is invoked to log out the target user.
4. If the HTTP code indicates an internal server issue on the client's `MessageReceiver`, `onError` is invoked, logging out the target user.
5. Finally a message is shown on the server's command line interface.

3.5 Client

The client consists of three parts.

- Main. Contains the entry point for the program
- MessageReceiver. Responsible for listening to request from the server and displaying it to the user interface
- MessageSender. Sends message entered on the user interface to the server for distribution

3.5.1 Main

The Main component is the entry point for the chat client. Its responsibilites include reading the user's command line for username and target host, starting up the MessageReceiver and polling its message channel, and finally processing user input and forwarding it to the MessageSender.

Implementation

The following code shows how the main component of the chat client operates.

```kotlin
fun main() {
    val userName = readUsername()
    val serverHost = readHostAddress()

    val sender = MessageSender(serverHost)
    val receiver = MessageReceiver(serverHost)

    runBlocking {
        val uuid = sender.login(UserData(userName)) ?: return@runBlocking

        launch {
            println("Start Receiver on Thread ${Thread.currentThread().name}")
            receiver.start()
        }

        launch {
            println("Start ChannelLoop on Thread ${Thread.currentThread().name}")
            val dateFormat = SimpleDateFormat("hh:mm:ss")
            for (m in receiver.channel) {
                println("(${dateFormat.format(Date())}) " +
                        "[${m.userName ?: m.sourceUser}] \"${m.message}\"")
            }
        }

        launch {
            println("Start MessageLoop on thread ${Thread.currentThread().name}")
            while (true) {
                delay(1000)
                val msg = readln()
                sender.send(MessageData(emptySet(), uuid, msg))
                if (msg == "exit") break
            }
            receiver.stop()
            sender.logout(LogoutData(uuid))
        }
    }
}

```

```kotlin
fun readHostAddress(): String {
    print("Enter Hostname: ")
    return readln()
}

fun readUsername(): String {
    print("Enter Username: ")
    return readln()
}
```

At the start of the main function username and target host address are read from the users command line using `readUsername()` and `readHostAddress()` and a `MessageSender` and `MessageReceiver` instance are created.

Inside `runBlocking`, a login request with the previously entered username as payload is sent to the server via `MessageSender::login`. If it returns null `runBlocking` is exited and subsequently the program execution ends. If the login succeeds, a UUID is received, which the client uses to uniquely identify themselves.

After the user is logged in, three coroutines are launched using the `launch` method.

1. The first coroutine starts the `MessageReceiver`.
2. The second coroutine begins polling `MessageReceiver`'s channel using a `for` loop. As long as this channel is not closed, the loop will print messages to the command line as soon as they are added to the channel.
3. The third coroutine begins by reading the user's command line for input, then uses `MessageSender::send` to send it to the server. The loop will end if the user inputs `"exit"`. The `MessageReceiver` will then stop, and the user will be logged out using `MessageSender::logout`.

3.5.2 MessageReceiver

The `MessageReceiver` is its own embedded server run on the client host and listening to one type of POST requests, which can only come from the KtorChat Server. It then deserializes those requests and puts them into a channel which gets polled in the main part of the client and then printed to the command line.

Implementation

The following code shows how the `MessageReceiver` operates.

```kotlin
class MessageReceiver(private val sourceHost: String) {
    val channel = Channel<MessageData>()
    private var server: JettyApplicationEngine? = null

    fun start() {
        server = embeddedServer(
            Jetty,
            port = Configuration.CLIENT_PORT,
            module = { module() })
        server?.start(wait = true)
    }

    fun stop() {
        server?.stop()
        channel.close()
    }

    private fun Application.module() {
        installPlugins()
        configureRoutes()
    }

    private fun Application.installPlugins() {
        // ...
    }

    private fun Application.configureRoutes() {
        // ...
    }
}
```

The MessageReceiver's constructor receives the address of the server where the messages are originating from as mandatory argument. When creating the instance of MessageReceiver the channel is started up.

Using start() an embedded server is created by calling embeddedServer(), which is a function supplied by the Ktor framework. This server is configured using the module() function, which is going to be explained later. Afterwards the server is started via server.start() with wait set to true, which means that the server is going to wait for incoming requests.

The MessageReceiver can be stopped via stop(), which will stop the embedded server and close down the channel.

Configuration of the embedded server is done via module(), which calls both installPlugins() and configureRoutes(). Note that those three functions are extension function for the class Application, which provides important functions like install() and routing().

installPlugins

The two plugins ContentNegotiation and ServerReadyCallback are installed to the embedded server within installPlugins().

```
private fun Application.installPlugins() {
    install(ContentNegotiation) {
        gson()
    }
    install(ServerReadyCallback) {
        callback {
            println("Receiver is running.")
        }
    }
}
```

- ContentNegotiation as seen in section 3.4.1
- ServerReadyCallback as seen in section 3.4.1

configureRoutes

The routes that specify how to access the embedded server are located inside the method configureRoutes.

```
private fun Application.configureRoutes() {
    routing {
        post("/receive") {
            val message = call.receive<MessageData>()
            if (call.request.local.remoteHost == sourceHost) {
                channel.send(message)
                call.respond(HttpStatusCode.OK)
            } else {
                call.respond(HttpStatusCode.Unauthorized)
            }
        }
    }
}
```

To define routes, `post()` is called inside `routing()`.

Inside `post`, the request's host address is compared to the host address specified at the initialization of `MessageReceiver` to ensure that the request came from the KtorChat server. If the two addresses do not match, `HttpCode 401 (Unauthorized)` is returned and message is ignored. If they match, `call.receive<MessageData>()` reads the text of the request and automatically deserializes it to a `MessageData` object before inserting it into the channel. `HttpCode 200 (OK)` is then returned in response to the request.

3.5.3 MessageSender

The MessageSender validates messages and login/logout POST requests and sends them to the KtorChat server. It also saves the host address of the KtorChat server to ensure that subsequent responses are from the server and not someone else.

Implementation

The following code shows how the `MessageSender` operates.

```
class MessageSender(private val serverHost: String) {
    private val client = HttpClient {
        install(ContentNegotiation) {
            gson()
        }
        defaultRequest {
            url {
                this.host = serverHost
                this.port = Configuration.SERVER_PORT
            }
        }
        install(HttpTimeout) {
            requestTimeoutMillis = 2000
        }
    }

    suspend fun send(messageData: MessageData) {
        // ...
    }

    suspend fun login(user: UserData): UUID? {
        // ...
    }

    suspend fun logout(logoutData: LogoutData) {
        // ...
    }
}
```

The MessageSender's constructor receives the address of the server where the messages are originating from as mandatory argument. When creating the instance of MessageReceiver a HttpClient instance is created and configured using the following plugins.

- ContentNegotiation is, as seen in section 3.4.1, responsible for automatically setting the correct Accept and Content-Type headers as well as serializing/deserializing the content in a specific format.
- DefaultRequest, as seen in section 3.4.3, is a plugin that sets the request's default parameters. In this case the target host and port are set automatically for all requests made with this instance.
- HttpTimeout as seen in section 3.4.3.

send

The method send sends a message post request to the server with a MessageData object as its body.

```
suspend fun send(messageData: MessageData) {
    client.post("message") {
        setBody(messageData)
    }
}
```

login

The method login sends a login post request to the server with a UserData object as its body.

```
suspend fun login(user: UserData): UUID? {
    val response: HttpResponse = try {
        client.post("login") {
            setBody(user)
        }
    } catch (_: HttpRequestTimeoutException) {
        println("Could not reach server!")
        return null
    }

    if (!response.status.isSuccess()) {
        println(response.body<String>())
        return null
    }

    val body: LoginResponse = response.body()
    println(body.errorMessage ?: "Logged In! -> $response")
    return body.id
}
```

If the server is not reachable, the request call will throw a HttpRequestTimeoutException and the method returns null. If the response status code is anything other than a 2xx HTTP Code, the procedure will deserialize the response as String and will also return null. If no exception were raised and the response was successful, the response body is deserialized to a LoginResponse object and the UUID identifying the user is returned.

logout

The method `logout` sends a `logout` post request to the server with a `LogoutData` object as its body.

```kotlin
suspend fun logout(logoutData: LogoutData) {
    val response = client.post("logout") {
        setBody(logoutData)
    }

    println("Logged out! -> $response")
}
```

4 Comparison to Spring

Spring Boot and Ktor are two modern frameworks for building web applications. Spring Boot, part of the larger Spring ecosystem, is designed for creating microservices with Java. Ktor is aimed at building asynchronous servers and clients in connected systems. Both offer robust features but employ to different programming languages and development styles.

4.1 Functional Criteria

4.1.1 Java vs Kotlin

Java and Kotlin are both popular languages in the development ecosystem, each with its own strengths, particularly in the context of web frameworks like Spring Boot and Ktor.

Java, the language behind Spring Boot, is known for its stability, extensive ecosystem, and vast community support. It's a language that has been around for decades (released in 1996 [7]), providing developers with a wealth of libraries and tools. Spring Boot leverages Java's robustness, offering a comprehensive framework that simplifies the development of web and enterprise applications with its convention-over-configuration approach.

Kotlin, released in 2016 [8] is a newer, more modern language designed to interoperate fully with Java while fixing some of Java's drawbacks, such as verbosity. Kotlin brings to the table a concise syntax, null safety, and functional programming features, making the development process more efficient and enjoyable. Ktor utilizes Kotlin's features to provide a lightweight, idiomatic way to build asynchronous web applications, appealing to developers looking for simplicity and expressiveness.

4.1.2 Route creation & Request Handling

In Spring Boot, route creation and request handling are managed through annotations like `@RestController` and `@RequestMapping`, where each function or method is mapped to a specific HTTP method. This approach leverages the Spring ecosystem, offering integrations with security, data handling, and more, albeit with overhead in terms of configuration and verbosity.

Ktor, on the other hand, adopts a more programmatic approach. Routes are defined in a functional style, where the routing structure can be nested and composed dynamically. This setup not only makes Ktor's routes easy to read and develop but also highly flexible, allowing developers to handle requests with direct access to the Kotlin language features such as coroutines for handling asynchronous operations. This leads to less boilerplate and a more straightforward way to implement complex request handling patterns.

Example

To process requests in Spring, a class annotated with `@RestController` has to be created. Within that class, methods that represent HTTP methods (`@GetMapping`, `@PostMapping`, and so on) are annotated to build single routes. The following example shows, how a simple route would look like:

```java
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello world!";
    }
}
```

The same route configured in Ktor would look something like the following:

```
fun main() {
    embeddedServer(Netty, port = 8080){
        routing {
            get("/hello") {
                call.respondText("Hello world!")
            }
        }
    }.start(wait = true)
}
```

4.1.3 Sendings Requests

Both Spring and Ktor offer user-friendly features for setting up HTTP clients, which are configurable in a multitude of ways.

Spring's `WebClient` is a reactive, non-blocking HTTP client part of the Spring Framework, designed for integration within the Spring ecosystem. It supports both synchronous and asynchronous operations using reactive streams API, making it ideal for applications requiring efficient handling of concurrent data flows. `WebClient` offers deep customization options through the Spring WebFlux framework, enhancing its utility in enterprise applications that demand robust, scalable solutions.

Ktor's `HttpClient` is built for Kotlin applications, utilizing Kotlin's coroutines for asynchronous programming, which aligns well with Kotlin's idiomatic style. It is highly modular and can be easily customized with features and plugins to extend functionality. This makes it particularly suitable for Kotlin-centric environments where developers value a lightweight, flexible client that integrates seamlessly with the overall application structure.

The purpose of the following code snippets is to demonstrate the fundamental operation of the two frameworks. Both of them have intricate configuration options that are too many to discuss in this context.

Instance creation

The following code snippets show, how Spring's `WebClient` and Ktor's `HttpClient` are created.

Spring

Spring employs the builder pattern. The builder can be accessed via the static function `builder` provided by the class `WebClient`. This builder then provides the function `baseUrl` which is used to set the target URL.

```
private final WebClient webClient = WebClient.builder().baseUrl("localhost:8080").build();
```

Ktor

As explained in the previous chapters, Ktor uses the constructor of the class `HttpClient` to create a new client. To set a base target URL, the plugin `DefaultRequest` is used.

```kotlin
private val httpClient = HttpClient {
    defaultRequest {
        url {
            host = "localhost"
            port = 8080
        }
    }
}
```

Both code snippets create a client, which is able to set off requests and sends them to `localhost:8080` by default.

Example 1: Ordinary request

The following code snippets show how to send a `get` request to an endpoint named "hello" and parse the result as a `String`.

Spring

```java
Mono<String> result = webClient.get()
    .uri("/hello")
    .retrieve()
    .bodyToMono(String.class);
String responseString = result.block();
```

Spring employs the builder pattern for creating requests. `get` specifies to use the HTTP methods GET. The method `uri` tells the client which path the request is sent to. The method `retrieve` sets off the request and `bodyToMono` deserializes the response body to the given class. Then, `Mono::block` waits for the request to be finished and returns the response body. This request would result in the underlying thread to be blocked, as `Mono::block` is blocking.

Ktor

To set off a get request using Ktor, the method `get` provided by an instance of `HttpClient` is used. The response body is then retrieved via the method `body` provided by the response object returned from the method `get`.

```kotlin
val response = httpClient.get("/hello")
val responseString = response.body<String>()
```

This request would not block the underlying thread, as all requests made with `HttpClient` are asychrounos by default.

Example 2: Request with retry

The following code snippets modify the previous examples to retry the request three times, when an error occurs.

Spring

To retry the request, the method `retry` can be added to chain invocations of the request builder. This method accepts an integer, which describes how many times the request should be retried.

```java
Mono<String> result = webClient.get()
    .uri("/hello")
    .retrieve()
    .bodyToMono(String.class)
    .retry(3);
String responseString = result.block();
```

Ktor

To retry the request, the plugin `HttpRequestRetry` is used. This plugin can be installed to single requests via the method `retry`.

```kotlin
val response = httpClient.get("/hello") {
    retry {
        maxRetries = 3
    }
}
val responseString = response.body<String>()
```

Example 3: Request with timeout

The following code snippets additionally modify the previous examples to also timeout if the request takes more than 2500 milliseconds.

Spring

To set a timeout for the request, the method `timeout` can be added to chain invocations of the request builder. This method accepts an instance of the class `Duration`.

```java
Mono<String> result = webClient.get()
    .uri("/hello")
    .retrieve()
    .bodyToMono(String.class)
    .retry(3)
    .timeout(Duration.ofMillis(2500));
String responseString = result.block();
```

Ktor

To set a timeout for the request, the plugin `HttpTimeout` is used. This plugin can be installed to single requests via the method `timeout`.

```kotlin
val response = httpClient.get("/hello") {
    retry {
        maxRetries = 3
    }
    timeout {
        requestTimeoutMillis = 2500
    }
}
val responseString = response.body<String>()
```

4.2 Non-Functional Criteria

In this section, a comparison with some non-functional criteria is done.

4.2.1 Community Support

Compared to more well-established frameworks like Spring Boot, Ktor has less third-party libraries and resources since its ecosystem and community are still expanding.

4.2.2 Ease of Use

Ktor has a shorter learning curve, especially for developers already familiar with Kotlin [9] [10]. Its design, centered around Kotlin coroutines and a focused API, contribute to an accessible entry point. On the other hand, Spring's learning curve can be steeper. The framework's comprehensiveness and heavy use of annotations, though powerful, might require more time for developers to get used to and master.

4.2.3 Performance

Ktor's non-blocking, asynchronous architecture allows it to support a large number of concurrent connections and is optimized for speed. Although Spring Boot provides great speed, it is not made primarily for asynchronous programming, therefore it might not be as effective when managing a large number of concurrent connections [11].

5 Conclusion

In this thesis the core principles and features of the asynchronous framework Ktor were explored. The example application KtorChat was intended to showcase as many features of both the server and the client part of the framework in a concise and easy to understand way as possible. Ktor uses Kotlin's language features to its fullest potential by making it incredibly performant using Coroutines and enabling developers to write simple yet effective code with the combination of multiple patterns, like lambda with receiver and trailing lambda.

Personally I enjoyed using Ktor. It does not require much configuration or setup and works right out-of-the-box the way you would expect it to. Compared to Spring and Java, the combination of Ktor and Kotlin felt much more intuitive and produced way more readable code with less boilerplate, thanks to Kotlin's many developer friendly features such as null safety, inherently declared getters and setters, primary constructors, and so on.

In conclusion, the choice between Ktor and Spring Boot for developing web applications depends significantly on the specific requirements and context of the project. Ktor, with its lightweight nature and asynchronous capabilities, is particularly well-suited for applications that require high performance and scalability in environments where resources are limited, such as in microservices architectures. Its streamlined setup and ease of use make it an excellent choice for projects where development productivity and application responsiveness are critical. Released in 2018 [12], Ktor is currently still regarded as an emerging framework with a small, but dedicated network of supporters. For new developers, the official documentation remains the sole reliable resource, but is of high quality.
On the other hand, Spring, which was first brought up in 2002 [13] is a robust framework with a vast ecosystem and a mature community, making it ideal for developers who require a comprehensive suite of features, extensive documentation, and community support. Its strength lies in its versatility and the vast array of integrated solutions it offers, which can significantly reduce development time and complexity in more extensive or more complex applications.
In summary, for projects needing rapid development and performance efficiency with simpler setup, Ktor is highly recommended. Conversely, for projects that benefit from a rich feature set and extensive community resources, Spring Boot poses as a compelling option.

References

[1] S. Sobral. CS1: C, Java or Python? Tips for a Conscious Choice. 12th annual International Conference of Education, Research and Innovation, Seville, Spain. 2019.

[2] Lambda Documentation. https://kotlinlang.org/docs/lambdas.html. Accessed: 2024-25-02.

[3] Coroutines. https://kotlinlang.org/docs/coroutines-overview.html. Accessed: 2024-25-02.

[4] Creating and configuring a client. https://ktor.io/docs/client-create-and-configure.html. Accessed: 2023-14-05.

[5] Creating a server. https://ktor.io/docs/server-create-and-configure.html. Accessed: 2023-14-05.

[6] Github Releases. https://github.com/ktorio/ktor/releases. Accessed: 2023-11-05.

[7] JDK Releases. https://www.java.com/releases/. Accessed: 2024-14-04.

[8] Kotlin 1.0 Released: Pragmatic Language for the JVM and Android. https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/. Accessed: 2024-14-04.

[9] Spring Boot vs. Ktor: A Battle of Kotlin-Powered Application Servers. https://dev.to/sergiomarcial/spring-boot-vs-ktor-a-battle-of-kotlin-powered-application-servers-2p29. Accessed: 2024-14-04.

[10] Ktor: The Next Generation Framework That Might Replace Spring Boot. https://medium.com/@chaewonkong/ktor-the-next-generation-framework-that-might-replace-spring-boot-868e8d21fc0f. Accessed: 2024-14-04.

[11] Comparison to Spring on Performance. https://medium.com/@chaewonkong/ktor-the-next-generation-framework-that-might-replace-spring-boot-868e8d21fc0f. Accessed: 2024-19-03.

[12] Ktor 1.0 Released: A Connected Applications Framework by JetBrains. https://blog.jetbrains.com/kotlin/2018/11/ktor-1-0/. Accessed: 2023-11-02.

[13] Rod Johnson. *Expert One-on-One J2EE Design and Development*. Wiley, 2002.