Futures and asynchronous programming in Scala

Campanelli Alessio (878170)   Gottardo Mario (879088)
Munarin Andrea (879607)

January 12, 2025

Abstract

This paper presents a new datatype called "Future"[1] in the Scala programming language. The Future datatype is designed to address the issue of asynchronous programming in Scala, allowing for the execution of concurrent tasks without blocking the main thread of execution. The paper describes the implementation of the Future datatype, including its key features and how they are used to achieve concurrency and asynchrony. The paper also discusses the performance benefits of using the Future datatype and provides examples of its use in real-world applications. The paper concludes by discussing future work and potential improvements to the Future datatype. Overall, the Future datatype is a valuable addition to the Scala programming language, making it easier for developers to write concurrent and asynchronous code.

Contents

1 Introduction

1.1 Synchronous Programming

Suppose we are writing a program that calls a function which takes some time to elaborate the result, like a heavy mathematical calculation or simply an HTTP call. In a synchronous environment all of the calculations are carried one after the other, so the program will need to wait until the function returns to continue with its tasks.

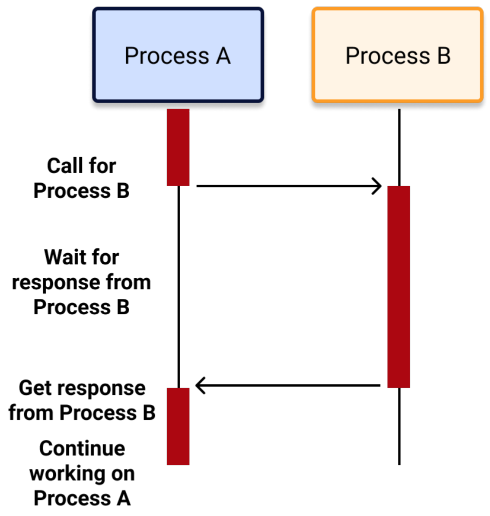
Process A
Process B
Call for Process B
Wait for response from Process B
Get response from Process B
Continue working on Process A

Figure 1: Synchronous processing

```
def createWebPage(): Unit =
  // long operations
  val data = fetchData()
  loadImages()
  loadTexts()
```

This is highly inefficient, since process A could carry out more operations concurrently that do not rely the result of process B.

```
[log] data fetched
[log] images loaded
[log] texts loaded
[info] total time: 1910ms
```

1.2 Asynchronous Programming

The same program can be refactored to work asynchronously, thanks to the use of futures.

```
def createWebPageAsync(): Unit =
  given ExecutionContext = ExecutionContext.global
  // long operations
  val data = Future[String]{fetchData()}
  val f1 = Future[Unit] {loadImages()}
  val f2 = Future[Unit] {loadTexts()}
  // delegation completed
```

This pattern enables the programmer to delegate the execution of the function to another thread while the calling one keeps executing.

```
[info] delegation completed time: 138ms
[log] texts loaded
[log] data fetched
[log] images loaded
[info] total time: 1302ms
```

The results are not surprising: we can see that the order of the operations is different from the order in which they are called (texts are loaded even before the data is fetched) and that the total execution time is reduced by 33%. The most interesting thing is the first line of output, which is the first instruction after the creation of the three futures. The fact that it's printed before the others shows the non-blocking nature of futures, that are executed independently of the workflow of the calling function.

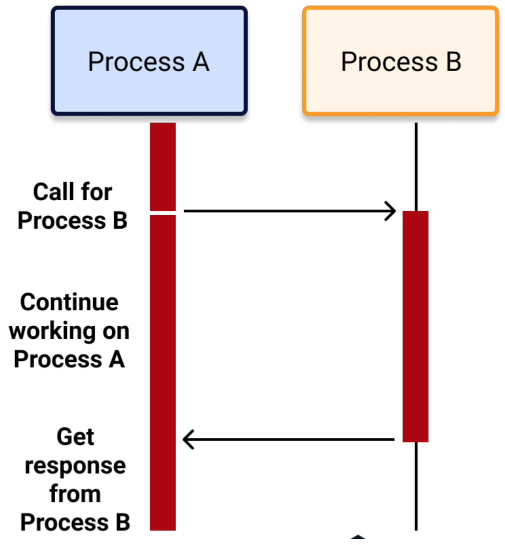

Figure 2: Asynchronous processing

2 The `Future` type

The Future datatype in Scala is a construct that represents a computation that may not have completed yet. It allows to perform asynchronously time-consuming or I/O bound tasks and handle their results in a non-blocking way.

The Future datatype is defined in the `scala.concurrent` package, and it has two main components:

- A computation that is executed asynchronously, which is passed as a parameter to the Future constructor.
- A `Promise` object, which is used to complete the Future with a value or an exception.

To create a Future, we can use the apply method of the Future companion object, which takes a by-name parameter (a function that returns a value or throws an exception) and returns a Future

```scala
import scala.concurrent._
import ExecutionContext.Implicits.global

val future: Future[Int] = Future {
  // some long running task
  Thread.sleep(1000)
  42
}
```

or can also use the `Future.successful` or `Future.failed` methods to create a completed Future with a value or an exception, respectively:

```scala
val successfulFuture: Future[Int] = Future.successful(42)
val failedFuture: Future[Int] = Future.failed(new Exception("something
    went wrong"))
```

The Future datatype has several methods to handle the result of the computation:

- `onComplete`: takes a callback function that will be executed when the Future completes, regardless of whether it completed successfully or with an exception.
- `foreach`: takes a callback function that will be executed when the Future completes successfully.
- `map` and `flatMap`: allow you to transform the value of the Future or compose multiple Futures, if it has been successful.

Here is an example of using the foreach method to handle the result of a Future:

```scala
val future: Future[Int] = Future {
  Thread.sleep(1000)
  42
}

future.foreach { result =>
  println(result) // prints 42 after 1 second
}
```

In this example, the foreach method is used to define a callback function that will be executed when the Future completes.

2.1 Execution context

An **execution context** in Scala is an abstraction that is used to execute code concurrently, in particular is an object that provides a thread pool and a way to execute tasks on that. The pool is used to run the computations represented by Future and Promise objects, and is required for their asynchronous operations.

The `ExecutionContext` trait has a single abstract method `execute(Runnable)` that takes a `Runnable` (the Java interface for thread creation) object and executes it asynchronously. The `ExecutionContext.Implicits.global` object is the default execution context and it is used when no other execution context is explicitly provided. It's backed by a `ForkJoinPool` (a thread pool that is optimized for parallelism) and it can be used for most general-purpose concurrent programming.

```
given ExecutionContext = ExecutionContext.global // without this the
    program would not compile
val future = Future[String] {
    // some long running task
    Thread.sleep(1000)
    "Task completed"
}

future.foreach { result =>
    println(result) // prints "Task completed" after 1 second
}
```

One can also provide their own execution context, for example, if there is the need to use a different thread pool with different parameters.

```
given ExecutionContext = ExecutionContext.fromExecutor(Executors.
    newFixedThreadPool(10))
...
```

The context is a parameter required by every method that can be applied on futures, since, as we will see later, their execution is asynchronous with respect to the main thread. Since the context parameter is declared as `implicit`, we can use the given keyword to define the context without passing it explicitly to the methods.

2.2 Callbacks

A ***callback*** or ***callback function*** is any reference to executable code that is passed as an argument to another piece of code; that code is expected to *call back* (execute) the callback function as part of its job. This execution may be immediate as in a synchronous callback, or it might happen at a later point in time as in an asynchronous one. Even though synchronous callbacks (in which we block our execution until the future provide the value we needed) are allowed by the Scala Future API as we will show later, from a performance point of view a better way to do it is in a completely non-blocking way.

If the Future has already been completed when registering the callback, then the callback may either be executed asynchronously, or sequentially on the same thread.

`onComplete[U](f: (Try[T]) => U): Unit`

This is the most important method related to callbacks. The callback function is automatically invoked when Future complete its work. The `onComplete` method wants as input a function: `Try[T] => U`. If the Future complete successfully, the callback is applied to the value of `Success[T]`, otherwise we have `Failure[T]`. Let's see an example of how this works:

```
import scala.util.{Success, Failure}

val f: Future[List[String]] = Future {
    session.getRecentPosts()
}

```

```scala
f.onComplete {
    case Success(posts) => for post <- posts do println(post)
    case Failure(t) => println("An error has occurred: " + t.
  getMessage)
}
```

`foreach[U](f: (T) => U): Unit`

This works similarly as `onComplete` but handles only successful values. `For-do` syntax simplifies the code:

```scala
val f: Future[List[String]] = Future {
    session.getRecentPosts()
}

for
  posts <- f
  post <- posts
do println(post)
```

Notice that both, `onComplete` and `foreach` methods has return type `Unit`. This design is intentional in order to **avoid chained invocations** because someone may think that they are processed in an ordered way. Indeed, callbacks registered on the same future are unordered.

2.2.1 Callback order

We have no precise information about when a callback is invoked and from what thread (may be different from the thread that execute the Future). The only thing we can say is that the callback is executed by some thread, at some time after the future object is completed. We say that the callback is executed ***eventually***. Another important concept is that, for what we said before, **different callbacks can be executed at the same time**, breaking the program logic and leading to errors.

```scala
@volatile var totalA = 0

val text = Future {
  "na" * 16 + "BATMAN!!!"
}

text.foreach { txt =>
  totalA += txt.count(_ == 'a')
}

text.foreach { txt =>
  totalA += txt.count(_ == 'A')
}
```

The expected value from the above code is 18, but since the two functions are executed simultaneously we can obtain 2 or 16 since "+=" **is not an atomic operation**. Notice also the importance of the `volatile` annotation. Reads and writes on variables marked as volatile are never reordered. If a write W to a volatile variable v is observed on another thread through a read R of the same variable, then all the writes that preceded the write W are guaranteed to be observed after the read R.

2.2.2 Summary

- **Registering an onComplete** callback on the future ensures that the corresponding closure is invoked after the future is completed, **eventually**.
- **Registering a foreach** callback has the same semantics as onComplete, with the difference that the closure is **only called** if the future is completed **successfully**.
- In the event that multiple callbacks are registered on the future, the **order in which they are executed is not defined**. In fact, the callbacks may be executed concurrently with one another. However, a particular ExecutionContext implementation may result in a well-defined order.
- If some callbacks throws an exception, the others are executed **regarderless**.
- In the event that **some callbacks never complete** (for examples, the callback contains an infinite loop), the other callbacks may not be executed at all. In these cases, a potentially blocking callback must use the blocking construct (see below).
- **Once executed**, the callbacks are removed from the future object, thus being **eligible for GC**.

2.3 Functional composition and for-comprehensions

The callback mechanism we have shown is sufficient to **chain Future results with subsequent computations**. This sometimes results in ***bulky code***. Here's an example, assuming that we have an API for interfacing with a currency trading service:

```
val rateQuote = Future {
  connection.getCurrentValue(USD)
}

for quote <- rateQuote do
  val purchase = Future {
    if isProfitable(quote) then connection.buy(amount, quote)
    else throw Exception("not profitable")
  }

  for amount <- purchase do
    println("Purchased " + amount + " USD")
```

This is inconvinent for two reason mainly:

- We have to use foreach and **nest** the second purchase Future within it. Imagine that after the purchase is completed, we want to sell some other currency.
- The `purchase` **Future is not in the scope** with the rest of the code.

`map[S](f: (T) => S): Future[S]`

One of the best ways to improve the code written before is to use the `map` function:

```
val rateQuote = Future {
  connection.getCurrentValue(USD)
}

val purchase = rateQuote.map { quote =>
  if isProfitable(quote) then connection.buy(amount, quote)
  else throw Exception("not profitable")
}

purchase.foreach { amount =>
  println("Purchased " + amount + " USD")
}
```

By using `map` on `rateQuote` we have eliminated one foreach callback and, more importantly, the nesting. Notice, also, how the `map` function propagate correctly unhandled exception.

`flatMap[S](f: (T) => Future[S]): Future[S]`

One of the design goals for futures was to enable their use in **for-comprehensions**. For this reason, futures also have the `flatMap` and `withFilter` combinators. Now we will provide an example using these two methods, introducing the exchange of US dollars for Swiss francs (CHF):

```
val usdQuote = Future { connection.getCurrentValue(USD) }
val chfQuote = Future { connection.getCurrentValue(CHF) }

val purchase = for
  usd <- usdQuote
  chf <- chfQuote
  if isProfitable(usd, chf)
yield connection.buy(amount, chf)

purchase.foreach { amount =>
  println("Purchased " + amount + " CHF")
}
```

As we saw in class, for-comprehensions is just a syntactic sugar of `flatMap` and `map` functions:

```
val purchase = usdQuote.flatMap {
  usd =>
    chfQuote
      .withFilter(chf => isProfitable(usd, chf))
      .map(chf => connection.buy(amount, chf))
}
```

The filter combinator creates a new Future which contains the value of the original future **only if it satisfies some predicate**. Otherwise, the new Future is failed with a `NoSuchElementException`.

Now we are going to see three very useful method that works similarly to `map` and `flatMap` with different purpose.

`def recover[U >: T](pf: PartialFunction[Throwable, U]): Future[U]`

Since the Future trait can conceptually contain two types of values (computation results and exceptions), there exists a need for combinators which handle exceptions. The keyword for this operation is `recover`:

```scala
val purchase: Future[Int] = rateQuote.map {
  quote => connection.buy(amount, quote)
}.recover {
  case QuoteChangedException() => 0
}
```

The `recover` methods returns the same value of the Future if it succeeds. Instead, if it captures the Throwable then it will return a new Future with the value computed as results.

`def fallbackTo[U >: T](that: Future[U]): Future[U]`

The `fallbackTo` method is used to create a new Future that will complete with the result of the first Future that completes. The **second Future is used as a fallback if the first one fails**. If the first Future completes successfully, the second Future will not be executed. The `fallbackTo` method is useful for providing a default value or alternative action in case the primary action fails.

```scala
val usdQuote = Future {
  connection.getCurrentValue(USD)
}.map {
  usd => "Value: " + usd + "$"
}
val chfQuote = Future {
  connection.getCurrentValue(CHF)
}.map {
  chf => "Value: " + chf + "CHF"
}

val anyQuote = usdQuote.fallbackTo(chfQuote)

anyQuote.foreach { println(_) }
```

`def andThen[U](pf: PartialFunction[Try[T], U]): Future[T]`

In the end the `andThen` combinator is used to **chain multiple Future objects together**. It allows you to specify a callback function that will be executed when the previous Future completes, and the result of that callback function is used to create a new Future object. This ensures that multiple `andThen` calls are ordered, as in the following example which stores the recent posts from a social network to a mutable set and then renders all the posts to the screen:

```scala
val allPosts = mutable.Set[String]()

Future {
  session.getRecentPosts()
}.andThen {
  case Success(posts) => allPosts ++= posts
}.andThen {
  case _ =>
    clearAll()
    for post <- allPosts do render(post)
}
```

All the combinators we have seen in this section represents the **purely functional style of Scala**. Indeed, every function returns a new Future without any side effect on existing objects.

2.4 Projections

Projections are very useful if we want to handle failed Future computation using the structure of for-comprehensions. If the original future fails, the failed projection returns a **Future containing a value of type Throwable**. If the original future succeeds, the failed projection fails with a `NoSuchElementException`:

```scala
val f = Future {
  2 / 0
}
for exc <- f.failed do println(exc) //print error

val g = Future {
  4 / 2
}
for exc <- g.failed do println(exc) //print nothing
```

`for-do` is just a syntactic sugar and in this case can be translated as:

```scala
f.failed.foreach(exc => println(exc))
```

2.5 Exceptions

When the computation inside a Future **throws an unhandled exception**, then our object will **fail**. Failed futures store an instance of Throwable instead of the result value. As we saw, Future provide the failed projection method: using this is possible to handle Throwable as a success of another Future. There exists some exception that are treaded differently:

- `scala.runtime.NonLocalReturnControl[_]`
  - Used to indicate that a **non-local return** has occurred from a function.

  - The class is used by the Scala compiler when generating code for a return statement that appears inside a nested function.
  - Instead of keeping this exception, the **associated value is stored into the future** or a promise.

- `ExecutionException`
  - Unhandled `InterruptedException`, `Error` or a `scala.util.control.ControlThrowable`
  - In this case, the `ExecutionException` has the unhandled exception as its cause
  - **To prevent propagation** of critical and control-flow related exceptions normally not handled by the client code and at the same time inform the client in which future the computation failed.

- *Fatal exceptions*
  - **Rethrown** in the thread executing the failed asynchronous computation
  - **Informs** the code managing the executing threads of the problem and allows it to fail fast, if necessary.

Example of `scala.runtime.NonLocalReturnControl`:

```scala
import scala.runtime.NonLocalReturnControl

def foo() = {
    def bar() = {
        throw new NonLocalReturnControl[Int](10)
    }
    try {
        bar()
    } catch {
        case ex: NonLocalReturnControl[Int] => ex.value
    }
}

val result = foo()
println(result) // prints "10"
```

3 Futures as Monads

Let's start by recalling what a Monad is. A monad can be defined as a wrapper that represents a value of a generic type `A` plus the effect of some computation on it.

Theorem 3.1. *Scala Future are Monads.*

Proof. In order to prove that Future are Monads, it must be shown that the type Future provides:

- a type constructor `Future[_]`

- a function `return: A -> Future[A]`
- a function `>>=: Future[A] -> (A -> M[B]) -> M[B]`

Moreover it must satisfy the three monadic laws:

- `return(a) >>= k = k(a)`
- `m >>= return = m`
- `m >>= k >>= h = m >>= (x => (k(x) >>= h))`

As we can see in the Future class documentation[2], it provides:

- the type constructor `Future{}`
- the unit functions `Success[A](a: A)` and `Failure(t: Throwable)`
- the flatMap function: `flatMap[S](f: (T) => Future[S]): Future[S]`

So we can move on to the three monadic laws:

First monadic law

```
Success(a).flatMap(k) = k(a)
Success(a).flatMap(k) =
    Success(a) match
        case Success(a) => k(a)
        case Failure(t) => Failure(t)
    = k(a)
```

Second monadic law

```
m.flatMap(x => Success(x)) = m
m.flatMap(x => Success(x)) =
    m match
        case Success(x) => Success(x)
        case Failure(x) => Failure(x)
    = m
```

Third monadic law

```
m.flatMap(k).flatMap(h) = m.flatMap(x => k(x).flatMap(h))
```

We have to consider the two possible cases of Future: `Success(x)` and `Failure(t)`

- `Success(x)`:

```
Success(x).flatMap(k).flatMap(h) = Success(x).flatMap(x => k(x).flatMap(h))
                 k(a).flatMap(h) = (x => k(x).flatMap(h))(a)
                 k(a).flatMap(h) = k(a).flatMap(h)
```

- `Failure(t)`

  ```
  Failure(t).flatMap(k).flatMap(h) = Failure(t).flatMap(x => k(x).flatMap(h))
                        Failure(t) = Failure(t)
  ```

  Since `Failure(t).flatMap(f) = Failure(t)` $\forall$ `f`

Thus we can state that Future is a Monad. $\square$

4 Blocking and Promises

4.1 Blocking

So far we have consider total asynchronous scenarios, in which we just provide callbacks to handle the results but never consider their relationship with the main thread. However, sometimes we have to wait from the main thread for the execution of a particular Future, so we are introducing a kind of synchronization. Synchronization can be achieved in two ways: by invoking arbitrary blocking code within the Future, or by blocking outside another Future, waiting for its completion.

Blocking inside a Future Achievable thanks to the `blocking` construct even if the implementation depends on the provided Execution Context.

```scala
def foo(x: Int, y: Int): Int = {
    Thread.sleep(2000)
    println("inside")
    x * y
}

given ExecutionContext = ExecutionContext.global

val f = Future {
    blocking {
          println("before")
          foo(2,3)
          println("waiting")
    }
    println("waited")
}
```

Output:

```
before
inside
waiting
waited
```

Note that, in case of exception in the blocking code, it is forwarded to the caller.

Blocking outside the Future Blocking on a future is a bad practice that may lead to deadlocks and lost of performance, we can say that it's nicer to handle future in a pure functional way, by using callbacks and combinators.

If it's necessary to block for a Future, a better practice is to wait outside, using the Future and Promise APIs. Thanks to this schema, we can set a timeout to prevent deadlocks. Indeed Future implements the Awaitable trait with two methods

- `ready[T](awaitable: Awaitable[T], atMost: Duration): awaitable.type`
  Simply waits for the completion of the awaitable parameter for atMost time without returing its result

- `result[T](awaitable: Awaitable[T], atMost: Duration): T`
  Also waits for the completion, but in addition returns the value of the awaitable.

As for the previous example, `ready` and `result` are called by the Execution context and not by the client.

```scala
val rateQuote = Future {
    Thread.sleep(1000)
    println("waiting")
    10000
}

val purchase = rateQuote.map{ quote =>
    if quote >= 1000 then println("buy now!")
    else throw Exception("not profitable")
}

Await.ready(purchase, 2.seconds)
println("waited")
```

Output:

```
waiting
buy now!
waited
```

4.2 Promises

Futures can also be created using Promises[3]. Informally, a Promise can be thought as a writable, single-assignable container which completes a future. It represent a value that will be ready in the future and has two states:

- Unfulfilled: the computation is in progress and the result is not ready.
- Fulfilled: the computation is completed and the result is available.

A Promise can complete a Future in two ways: success (with a value) or failure (with an exception). A promise can be wrote a single time, indeed, trying to overwrite a yet fulfilled promise will result in an IllegalStateException. Here's a simple example:

```scala
val p = Promise[Int]()

def foo(): Unit = {
    Thread.sleep(1000)
    val r = Random.nextInt()
    if r % 2 == 0 then
        p.success(r)
    else
        p.failure(Exception("r is not even"))
}

// loaded on scala
scala> p
val res1: scala.concurrent.Promise[Int] = Future(<not completed>)

// foo called
scala> foo()

// after 1 second
scala> p
val res2: scala.concurrent.Promise[Int] = Future(Success(1081897598))

// or
scala> p
val res3: scala.concurrent.Promise[Int] = Future(Failure(java.lang.
    Exception: r is not even))
```

Moreover, a Promise can also be completed using the complete method, which takes a `Try[T]` monad type. This method is complementary to the two previous methods `Promise.success` and `Promise.failure`.

At a first look, the concepts of Promise and Futures seems to be very similar, almost overlapping. Indeed, they are strongly related but serve asynchronous programming in two different ways. While a Future is a read only representation of a computation, a Promise is writable representation of a computation. From a Promise object we can also extract the corresponding Future object, in order to provide the known handlers like `onComplete`.

5 Real case scenario

In the real world, all this environment fits the current cloud services architectural style, based on web servers providing services through protocols like http in so called APIs.

To better understand what we have discussed for now, let's build a simple function to provide a command line utility that, given a city name, asks to a service the current weather conditions and returns the current temperature. To do so we'll use the Open-Weather API[4].

The goal is to provide a simple high level function:

```
def getTemp(city: String): Future[Double]
```

First of all we need a function to get the coordinates of the desired city:

```
def getCoordinates(city: String): Future[(Double, Double)] = {
    basicRequest.get(
        uri"http://api.openweathermap.org/geo/1.0/direct?q=$city&appid
    =$apiKey")
        .send(HttpClientSyncBackend()).body
    match
```

```
        case Left(error) => Future.failed(Exception(error))
        case Right(body) =>
            val json = ujson.read(body)
            if json.arr.isEmpty then
                Future.failed(Exception("city does not exist"))
            else Future.successful((json(0)("lat").num, json(0)("lon")
    .num))
}
```

The second required function gets the current temperature of a particular coordinate.

```
def getWeather(lat: Double)(lon: Double): Future[Double] = {
    basicRequest.get(
        uri"https://api.openweathermap.org/data/2.5/weather?lat=$lat&
    lon=$lon&units=metric&appid=$apiKey")
        .send(HttpClientSyncBackend()).body
    match
        case Left(error) => Future.failed(Exception(error))
        case Right(body) => Future.successful(ujson.read(body)("main")
    ("temp").num)
}
```

And so we can implement the `getTemp` as follows:

```
def getTemp(city: String): Future[Double] = {
    val apiKey = "<API-KEY>"

    def getCoordinates(city: String): Future[(Double, Double)] = {
        ...
    }

    def getWeather(lat: Double)(lon: Double): Future[Double] = {
        ...
    }

    getCoordinates(city)
        .flatMap((lat, lon) => getWeather(lat)(lon))
}
```

The function can be used in this way:

```
val tempFuture = getTemp("Venezia")
tempFuture.onComplete(temp => temp match
    case Failure(exception) => println("error: " + exception)
    case Success(value) => println(value)
)

Await.ready(tempFuture, 3.second)}
```

6 Conclusions

In conclusion, the Future monad, combined with the blocking pattern and promises, offers a great functional suite to achieve asynchronous programming. Futures and promises are a great abstraction of multithreading, highly reducing the verbosity and the complexity of other languages like Java.

References

[1] Scala Future Documentation. `https://docs.scala-lang.org/overviews/core/futures.html`. [Online; accessed 19-January-2023].

[2] Scala Standard Library Future. `https://www.scala-lang.org/api/current/scala/concurrent/Future.html`. [Online; accessed 19-January-2023].

[3] Scala Standard Library Promise. `https://www.scala-lang.org/api/current/scala/concurrent/Promise.html`. [Online; accessed 24-January-2023].

[4] OpenWeather API documentation. `https://openweathermap.org/api`. [Online; accessed 24-January-2023].