Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea Magistrale in Informatica

The advantages of the Scala language in the context of Infrastructure as code

Master Thesis

Supervisor
Prof.ssa Crafa Silvia

Graduating
Cisotto Emanuele

Academic Year 2022-2023

Cisotto Emanuele: *The advantages of the Scala language in the context of Infrastructure as code,* Master Thesis, © April 2023.

Dedicato alla mia famiglia e ai miei amici

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di *stage*, della durata di circa trecentoventi ore, dal laureando Alessandro Rizzo presso l'azienda Infocert S.p.A. Gli obiettivi da raggiungere erano molteplici.
In primo luogo era richiesto lo studio e la comprensione dei fondamenti della *Chaos Engineering.*

In secondo luogo era richiesta l'implementazione di una versione rivisitata di un *software* aziendale già esistente, MICO, in seguendo i principi dell'architettura a microservizi e reactive tramite il *framework* Akka. Tale *framework* permette di utilizzare il modello ad attori per gestire il completamento di diversi task simultaneamente e in maniera asincrona. In questo sviluppo andava applicato quanto appreso nella fase di studio per progettare e realizzare un'applicazione il più resiliente possibile.

Infine, una volta completato lo sviluppo, andavano applicati tutti i principi di *Chaos Engineering* appresi durante la fase di studio per aumentare la fiducia nell'applicazione e per scoprire eventuali vulnerabilità non ancora considerate con lo scopo ultimo di aumentare la resilienza e l'affidabilità del prodotto.

"Failures are a given, and everything will eventually fail over time"

— Werner Vogels

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine alla Prof.ssa Silvia Crafa , relatrice della mia tesi, per la disponibilità e la cortesia mostratami durante quest'ultima parte del mio percorso.

Desidero ringraziare Tiziano Campili che mi ha assistito durante lo stage aziendale, in particolare per la sua disponibilità.

Voglio ringraziare infine la mia famiglia e tutti i miei amici che mi sono stati vicini durante questo percorso accademico consigliandomi sempre per il meglio e senza lasciarmi mai solo.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduction to Infrastructure as Code

An introduction to iac

1.1 Infrastructure as Code

1.1.1 What is IaC

Infrastructure as Code is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.
The IT infrastructure managed by this process comprises both physical equipment, such as bare-metal servers, as well as virtual machines, and associated configuration resources.
The definitions may be in a version control system. The code in the definition files may use either scripts or declarative definitions, rather than maintaining the code through manual processes, but IaC more often employs declarative approaches.

Types of approaches

There are generally two approaches to IaC: declarative (functional) vs imperative (procedural). The difference between the declarative and the imperative approach is essentially *what* versus *how*. The declarative approach focuses on what the eventual target configuration should be; the imperative focuses on how the infrastructure is to be changed to meet this. The declarative approach defines the desired state and the system executes what needs to happen to achieve that desired state. Imperative defines specific commands that need to be executed in the appropriate order to end with the desired conclusion.

Methods

There are two methods of IaC: push and pull. The main difference is the manner in which the servers are told how to be configured. In the pull method, the server to be configured will pull its configuration from the controlling server. In the push method, the controlling server pushes the configuration to the destination system.

1.1.2 Advantages of IaC

The value of IaC can be broken down into three measurable categories: **cost**, **speed**, and **risk**. Cost reduction aims at helping not only the enterprise financially, but also in terms of people and effort, meaning that by removing the manual component, people are able to refocus their efforts on other enterprise tasks. Infrastructure automation enables speed through faster execution when configuring your infrastructure and aims at providing visibility to help other teams across the enterprise work quickly and more efficiently. Automation removes the risk associated with human error, like manual misconfiguration; removing this can decrease downtime and increase reliability.
Related to the risk, we could highlight the importance of the **consistency** of such an approach. Through the manual modifications to the infrastructure achieved in a solution without IaC, at some point will be extremely hard to reproduce an exact configuration since some ad-hoc steps were required whilst some others were executed in a different order. Infrastructure as Code enforces consistency by allowing users to represent infrastructure environments using code. Therefore, the deployment and modification of resources will always be consistent and idempotent (i.e. every time a specific operation gets executed, the same result will be generated).
Furthermore, IaC tools usually offer mechanisms to enhance reusability. This feature makes your code base less verbose and more readable while at the same time team members are encouraged to apply best practices.
Finally, another big advantage of IaC is collaboration. Since the infrastructure resources are defined in configuration files it means that these files can be version controlled. At any given time, the team is able to collaborate together in order to modify an environment and even be able to see the history (from commits) of an infrastructure resource. This also makes debugging much easier and accurate.

1.1.3 Challanges of IaC

While IaC offers numerous benefits, there are also several challenges that organizations must address when implementing this approach.
One of the major challenges is the adoption discrepancies that arise when integrating new frameworks with existing technology. This requires careful coordination with other teams, particularly those responsible for security and compliance, and can result in difficulties in determining where resources are being delivered, controlled, and managed. To address these issues, organizations must continually communicate and audit their IaC adoption to minimize infrastructure drift and ensure that security measures remain up to date.

Another challenge is the need for security assessment tools that can effectively evaluate the dynamic nature of IaC. Traditional security measures may require significant cycles to be integrated with IaC, and there may be a need for human checks to ensure that resources are operating correctly and being used by the appropriate applications.

Organizations may need to invest in new tools or capabilities to ensure proper control and monitoring.

The implementation of IaC also requires a high degree of technical competence, which can result in the need for new human capital. Senior executives may face challenges in continually investing in employee skills, particularly if the organization is in the early adoption phase. Outsourcing IaC services may be a viable option for organizations to improve automation processes in terms of cost and overall IT infrastructure quality.

Versioning and traceability of settings can also be a challenge when IaC is utilized widely across an organization with various teams. As IaC becomes more complex, it can be difficult to keep track of infrastructure and identify infra-drift, making it essential to implement effective version control and tracking mechanisms.

1.1.4 Evolution of IaC

Tools designed for Serverless Applications - the first wave

The foundation of IaC in the public clouds is these three cloud vendor-specific tools: AWS CloudFormation in AWS, Azure Resource Manager (ARM) in Azure, and Cloud Deployment Manager in GCP. These are YAML or JSON based, declarative tools and have been in cloud toolboxes for a long time and require a fair amount of markup code. Tools with shortcuts or “conventions over configuration” were developed to boost productivity and make distributed microservice applications seem more like a traditional monolithic application or a framework. These tools provide best-practice defaults and enable building and testing your serverless applications locally on your machine.

Serverless Infrastructure as programming language code - the second wave

Declarative language has some limitations when there is the need to do more complex business logic than what parameters, conditions, mappings, and loops (Terraform only) allow to do. Sometimes, there is the need to use external scripting to have the work done. A programming language could address such a problem and let us get around these boundaries and limitations. This second generation tools generate the declarative markup code with the aid of a programming language, or bypass it and utilizes cloud APIs. These kind of tools with programming language support is a rising and trending approach in IaC at the moment.

Capitolo 2

Pulumi, an IaC platform

An introduction to Pulumi

2.1 Introduction to Pulumi

Pulumi is a cloud engineering platform that enables developers and infrastructure teams to build, deploy, and manage cloud-native applications and infrastructure across multiple cloud providers, including AWS, Azure, Google Cloud, and Kubernetes.
Pulumi provides a programming model that allows developers to use familiar languages, such as Python, JavaScript, TypeScript, Go, and (partially) Java to define their IaC and manage it as software. In fact, it belongs to the second generation tools of the IaC. As already mentioned in the Introduction to Infrastructure as Code chapter, such an approach, makes it easier to automate the deployment and management of infrastructure and applications, as well as to collaborate across teams and projects.
Pulumi offers a range of tools and features to simplify the development and management of cloud infrastructure, including version control, testing, continuous integration and delivery (CI/CD), monitoring, and security. It also provides templates, examples, and libraries for common infrastructure patterns and services, such as containers, serverless functions, databases, and networking.
Overall, Pulumi aims to streamline the process of building and managing modern cloud-native applications and infrastructure, while providing a flexible and developer-friendly experience.

2.1.1 The great advantages of Pulumi as a second generation IaC tool

First of all, as mentioned in the Serverless Infrastructure as programming language code - the second wave paragraph, all the functionalities that comes along a programming language are letting us achieve more robust and powerful solutions for our infrastructure, rather than what we could achieve with the expressive power of a markup language (like the ones used with Terraform). Being the focus of this thesis, we'll discuss more about such advantages in the Comparison between the languages for Pulumi and the advantages of Scala Chapter.

Furthermore, as aforementioned, Pulumi is a multi-cloud tool. Thanks to this we can rely on a single IaC tool for managing resources across different cloud platforms.
Moreover, Pulumi lets the user choose its favorite programming language, or the one that in its opinion is a best-fit for the need to be addressed. In other words, such a choice can both reduce the requirements placed on the user's knowledge, since it can choose among many different programming languages, and at the same time offer different programming paradigms to choose from, so that for any need there is a programming language that is addressing such a need better than the others.
Finally, Pulumi comes with a range of integrated tools and features, such as automatic parallelism, drift detection, and stack references, making it easier to manage complex infrastructure and deployments.

2.2 Pulumi functioning

2.2.1 The stack

2.2.2 APIs to define the resources to be created

2.2.3 Creating and updating resources with Pulumi commands

2.2.4 Viewing resources state

2.2.5 Restoring resources state

Capitolo 3

Scala, a modern functional and object-oriented programming language

Scala brief ovreview

3.1 Introduction to Scala

3.1.1 Scala, a modern functional and object oriented programming language

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It seamlessly integrates features of object-oriented and functional languages.

3.1.2 Object-orientation

Scala is a pure object-oriented language in the sense that every value is an object. Types and behaviors of objects are described by classes and traits. Classes can be extended by subclassing, and by using a flexible mixin-based composition mechanism as a clean replacement for multiple inheritance.

3.1.3 Functional paradigm

Scala is also a functional language in the sense that every function is a value. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and it supports currying. Scala's case classes and its built-in support for pattern matching provide the functionality of algebraic types, which are used in many functional languages. Singleton objects provide a convenient way to group functions that aren't members of a class.

3.1.4 Scala functionalities related to the work of the thesis

Higher order functions and currying

Thanks to the functional paradigm of Scala, I used the higher order functions and the currying feature to create a powerful syntactic sugar for the declaration of AWS EC2 IaC resources in a concise and elegant way.

Higher order functions An higher order function is function that either takes one or more functions as arguments or returns a function as return value, or both of them. Such a feature is at the basis of the functional programming paradigm since it lets concatenates function and create more complex and more powerful abstract constructs such as functors, applicatives and monads, so that we can eventually work on data in an immutable, and then safe, way. We'll see more about functors and monads in the Functor and monads paragraph.

Currying Currying is the transformation of a function with multiple arguments into a sequence of single-argument functions. That means, converting a function like `f(a, b, c, ...)` into a function like `f(a)(b)(c)...`. To give a more detailed example let's consider the following function in Scala:

```
def sum(a: Int, b: Int) : Int =
  a + b
```

Such a function, takes two Int parameters as input and returns the sum of them. Therefore, if we call `sum(1, 2)` we get `3` as result. Now let's instead consider the following function:

```
def csum(a: Int)(b: Int) : Int =
  a + b
```

Here we can call the function writing `csum(1)(2)` and we'll get 3, but we could also write `csum(1)` and get, as output, a function that takes a single `Int` in input and sums it to 1 (the `Int` we passed to `csum` previously). The returned function will be analogous to this function:

```
def csum1(b: Int) : Int =
  1 + b
```

So the currying is letting us define and use partial functions, and this feature, combined with the feature of taking functions as parameters, will be a key ingredient for the highly expressive and readable solution achieved and proposed in the Study case: AWS EC2 resource generation with Pulumi chapter.

Traits

Pattern matching

Pattern matching is a mechanism for checking a value against a pattern. A successful match can also deconstruct a value into its constituent parts. It is a more powerful version of the switch statement in Java and it can likewise be used in place of a series of if/else statements. Let's now consider the following example, taken from Tour of Scala - Pattern Matching, to have a better idea of the expressiveness of such a feature:

```
sealed trait Device
case class Phone(model: String) extends Device:
  def screenOff = "Turning screen off"

case class Computer(model: String) extends Device:
  def screenSaverOn = "Turning screen saver on..."


def goIdle(device: Device): String = device match
  case p: Phone => p.screenOff
  case c: Computer => c.screenSaverOn
```

We can notice how we are applying the pattern matching on the variable `device` to have a different behavior on the base of its actual type. This is useful when the case needs to call a method on the pattern. In fact, we'll see in the Study case: AWS EC2 resource generation with Pulumi chapter how this feature of Scala will be used to achieve our solution.
Obviously, in such an example, the right-hand side of the various cases must have a valid return type with respect to the return type given in the declaration of the method, that in this case is `String`.

Extension methods

Extension methods let you add methods to a type after the type is defined, i.e., they let you add new methods to closed classes. Let's consider an example from Scala 3 Book - Extension Methods about the calculation of the circumference of a circle.
In a file we may have:

```
case class Circle(x: Double, y: Double, radius: Double)
```

And in another:

```
extension (c: Circle)
def circumference: Double = c.radius * math.Pi * 2
```

Then we could have this code in our main method:

```
val aCircle = Circle(2, 3, 5)
aCircle.circumference
```

We shall notice that such extension methods are letting us extend types without relying on helper classes, letting us elegantly invoke methods on instances of the closed classes or objects we are referring to.

Given and using keywords

`given` has different usages, but we are interested into its capability to automatically construct an instance of a certain type and make it available to contexts in which we are expecting an implicit parameter. To mark a parameter as implicit we have to use the keyword `using`. Doing so, a `given` variable with a matching type to the implicit parameter's one, if present in the scope, will be automatically injected in **at compile time**.
Lets consider an example taken from Given and Using Clauses in Scala 3 - Rock the JVM to better understand such concepts:

```
given personOrdering: Ordering[Person] with {
  override def compare(x: Person, y: Person): Int =
    x.surname.compareTo(y.surname)
}
```

Here we are creating an instance of `Ordering` for the `Person` type. Now lets consider the following function declaration:

```
def listPeople(persons: Seq[Person])(using ordering: Ordering[Person]) = ...
```

We can notice how the `ordering` parameter has been marked with the `using` keyword. Now, when we'll have to call such a function, it'll require us to just pass the `persons` parameter, since the implicit one (`ordering`) will be automatically injected:

```
// the compiler will inject the ordering at the end of following function call
listPeople(List(Person("Weasley", "Ron", 15), Person("Potter", "Harry", 15)))
```

This is a key functionality that, among with other Scala features, will let us achieve our goal in our thesis as we will see in the Comparison between the languages for Pulumi and the advantages of Scala chapter.

Functors and monads

Functors A Functor for a type provides the ability for its values to be ”mapped over”, i.e., apply a function that transforms the value contained in a given context while remembering its shape. We can represent all types that can be ”mapped over” with `F`. F it's a type constructor: the type of its values becomes concrete when provided a type argument. Therefore we write it `F[_]`, hinting that the type F takes another type as argument. The definition of a generic `Functor` would thus be written as:

```
trait Functor[F[_]]:
  extension [A](x: F[A])
    def map[B](f: A => B): F[B]
```

The instance `Functor` to `List` is:

```
given Functor[List] with
```

```
    extension [A](xs: List[A])
      def map[B](f: A => B): List[B] =
        xs.map(f)
```

Here we can notice, as previously mentioned in the Given and using keywords paragraph, the `given` keyword is letting us create an instance of the Functor class for the `List` type.
Lets consider now the following usage of the `map` extension method on a list:

```
  val l: List[Int] = List(1, 2, 3)
  l.map(x => x  * 2) // the output is List(2, 4, 6)
```

The map method is now directly used on l. It is available as an extension method since `l`'s type is List[Int] and a given instance for Functor[List], which defines map, is in scope (thanks to the `given` keyword).

Monads A `Monad` provides the ability to sequence operations on values of a given type while maintaining the context of each operation. It is a generalization of the `Functor` concept, which allows us to apply a function to a value in a context. Such a generalization is letting us achieve a new level of expressiveness, that can be summarized as the chance to chain operations on a monadic value.
Like `Functors`, `Monads` can be represented by a type constructor `M[_]` that takes another type as an argument. The definition of a Monad is typically given in terms of two operations: `pure`, which lifts a value into the monadic context, and `flatMap`, which sequences operations on values in the context.
A generic Monad can be defined as follows:

```
  trait Monad[M[_]] extends Functor[M] {
    def pure[A](a: A): M[A]
    extension [A, B](ma: M[A])
      def flatMap[B](f: A => M[B]): M[B]
  }
```

And if we want to instantiate a `given` instance for the `List Monad` we shall write:

```
  given Monad[List] with {
    def pure[A](a: A): List[A] = List(a)
    extension [A, B](xs: List[A])
      def flatMap[B](f: A => List[B]): List[B] = xs.flatMap(f)
  }
```

To conclude consider this example on Lists:

```
  val xs = List(1, 2, 3)
  val ys = List(4, 5, 6)
  xs.flatMap(x => ys.map(y => x + y)) // List(5, 6, 7, 6, 7, 8, 7, 8, 9)
```

The fact we got a List[Int] as return type from the flatMap operation is letting us the chance to apply immediately after another operation on such a value. Differently, if we use map on xs, we will obtain the following result:

```
val xs = List(1, 2, 3)
val ys = List(4, 5, 6)
xs.map(x => ys.map(y => x + y)) // List(List(5, 6, 7), List(6, 7, 8), List(7, 8, 9))
```

that is of type List[List(Int)]. With this new type, we might have troubles in chaining operations since the data structure has changed.

For comprehension

Scala offers a lightweight notation for expressing sequence comprehensions. Comprehensions have the form `for (enumerators) yield e`, where `enumerators` refers to a semicolon-separated list of enumerators. An `enumerator` is either a generator which introduces new variables, or it is a filter. A comprehension evaluates the body `e` for each binding generated by the `enumerators` and returns a sequence of these values.
The for yield construct in Scala requires two functions to be defined on the type we are iterating on to work: `map`, and `flatMap`. In fact, it is not a coincidence that we previously introduced the concept of `Functor` and `Monad` and `map` and `flatMap` functions. To better *comprehend* how the for comprehension concept works lets consider a brief example:

```
val xs = List("foo", "bar", "baz")
val ys = List("hello", "world")

for {
  x <- xs
  y <- ys
} yield s"$x $y"
// List("foo hello", "foo world", "bar hello", "bar world", "baz hello", "baz world")
```

We can notice how we are iterating on the two lists to generate a List of string as result, made of the combinations of the two lists' elements.
The for yield construct, in combination with the `Monads`, will be a key feature to improve our Scala syntactic sugar for the Pulumi APIs.

Capitolo 4

Study case: AWS EC2 resource generation with Pulumi

Generation of AWS EC2 resources with Pulumi to compare how various languages supported by Pulumi will differ in the infrastructure resources declaration

4.1 Amazon Web Services

AWS is a wide collection of services with many different purposes and characteristics including compute, storage, databases, analytics, networking, mobile, developer tools, management tools, IoT, security, and enterprise applications: on-demand, available in seconds, with pay-as-you-go pricing. Anyway, for the purpose of the thesis we'll focus only on the EC2 module.

4.1.1 AWS's EC2 module

EC2 provides scalable computing capacity in the Amazon Web Services (AWS) Cloud. Amazon EC2 eliminates the need to invest in hardware up front, so that the development and deployment of the applications is faster. Such a characteristics is a perfect fit for an IaC scenario.

4.2 Case study infrastructure overview

For the thesis, only few components of the vast EC2 module have been selected to create a working infrastructure.
The infrastructure

4.2.1 Components of the infrastructure

VPC

Subnet

InternetGateway

RouteTable

4.3 Typescript implementation of the case study

4.4 Java implementation of the case study

4.5 My Scala implementation of the case study

4.5.1 Syntactic sugar

Syntactic sugar for the builders

Syntactic sugar for the constructors of the resources

Functor and Monads for the Output type

Usage of the syntactic sugar

4.5.2 Automatic code generation for the syntactic sugar

What Pulumi asks for the official support of a new Language

Raw automatic code generation with JavaParser

Usage of the generated code as a library

Possible future improvements for the code generation with Scalameta

Capitolo 5

Comparison between the languages for Pulumi and the advantages of Scala

Here all the advantages detected while using Scala for our study case will be presented, along with further considerations and comparisons about the pros and the cons in using a language instead of another one.

5.1 Typescript solution observations

5.1.1 Advantages of using Pulumi's Typescript APIs

Quite readable code

5.1.2 Disadvantages of using Pulumi's Typescript APIs

Poor tools to act on groups of Outputs

5.2 Java solution observations

5.2.1 Advantages of using Pulumi's Java APIs

5.2.2 Disadvantages of using Pulumi's Java APIs

Verbose code

5.3 My Scala solution observations

5.3.1 Advantages of using my Scala syntactic sugar

Very concise code

Very readable and elegant code

More powerful constructs thanks to the for-comprehension and the monads to act on groups of Outputs

5.3.2 Disadvantages of using my Scala syntactic sugar

Partial solution, not all the corner cases have been considered

5.4 Final thoughts on the Scala solution

Bibliography

Consulted web sites

10 years of cloud infrastructure as code - history and trends. URL: https://www.nordhero.com/posts/10-years-iac/.

Currying. URL: https://towardsdatascience.com/what-is-currying-in-programming-56fd57103431.

For Comprehension. URL: https://docs.scala-lang.org/tour/for-comprehensions.html.

Functors and Monads. URL: https://docs.scala-lang.org/scala3/reference/contextual/type-classes.html.

Infrastructure as Code. URL: https://en.wikipedia.org/wiki/Infrastructure_as_code.

Infrastructure as Code - Managing infrastructure resources with code. URL: https://towardsdatascience.com/infrastructure-as-code-f153d810428b.

Pattern matching. URL: https://docs.scala-lang.org/tour/pattern-matching.html#.

Pulumi. URL: https://www.pulumi.com/.

Using and Given. URL: https://blog.rockthejvm.com/scala-3-given-using/.