

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

1	Introduction to Infrastructure as Code	1
1.1	Infrastructure as Code	1
1.1.1	What is IaC	1
	Types of approaches	1
	Methods	2
1.1.2	Advantages of IaC	2
1.1.3	Challanges of IaC	2
1.1.4	Evolution of IaC	3
	Tools designed for Serverless Applications - the first wave . . .	3
	Serverless Infrastructure as programming language code - the second wave	3
2	Pulumi, an IaC platform	5
2.1	Introduction to Pulumi	5
2.1.1	The great advantages of Pulumi as a second generation IaC tool	5
2.2	Pulumi functioning	6
2.2.1	The stack	6
2.2.2	APIs to define the resources to be created	6
2.2.3	Creating and updating resources with Pulumi commands . . .	6
2.2.4	Viewing resources state	7
2.2.5	Restoring resources state	7
2.3	Pulumi's Output	7
3	Scala, a modern functional and object-oriented programming language	9
3.1	Introduction to Scala	9
3.1.1	Scala, a modern functional and object oriented programming language	9
3.1.2	Object-orientation	9
3.1.3	Functional paradigm	9
3.1.4	Scala functionalities related to the work of the thesis	10
	Higher order functions and currying	10
	Pattern matching	11

	Extension methods	11
	Given and using keywords	12
	Traits	12
	Functors and monads	13
	For comprehension	14
	Union type	15
4	Study case: AWS EC2 resource generation with Pulumi	17
4.1	Amazon Web Services	17
4.1.1	AWS's EC2 module	17
4.2	Case study infrastructure overview	17
4.2.1	Components of the infrastructure	18
	VPC	19
	Subnet	19
	InternetGateway	20
	RouteTable	20
4.3	Typescript implementation of the case study	20
4.3.1	VPC resource creation	20
4.3.2	Internet gateway creation	21
4.3.3	Subnets creation	21
	pulumi.all	22
	.apply	22
4.3.4	Routing table creation	23
4.3.5	Attaching the public subnets to the internet gateway	24
4.4	Creating the resources with Pulumi	24
4.5	Destroying the resources with Pulumi	26
4.6	My Scala implementation of the case study	27
4.6.1	Structure of the Java APIs for the constructors of the resources in Pulumi	27
4.6.2	Syntactic sugar usage	28
	Vpc creation	28
	Internet gateway creation	28
	Routing table creation	29
	Subnets creation	29
	Attaching the subnets to the routing table	30
4.6.3	Syntactic sugar for the constructors of the resources	30
	Correspondence between the input parameters and the user defined code used to create the VPC	31
	Vpc construction	32
	The baseOpts function	32
4.6.4	Syntactic sugar for the builders' methods	33
	Implicit conversion functions	36
4.6.5	Functor and Monad implementation for the Output type	37

Functor implementation	37
Monad implementation	38
4.6.6 Automatic code generation for the syntactic sugar	39
Pulumi Java APIs libraries inspection with JavaParser	39
Raw automatic syntactic sugar code generation	40
5 Comparison between the languages for Pulumi and the advantages of Scala	43
5.1 Typescript solution observations	44
5.1.1 Advantages of using Pulumi's Typescript APIs	44
Quite readable code	44
5.1.2 Disadvantages of using Pulumi's Typescript APIs	44
Poor tools to act on groups of Outputs	44
5.2 Java solution observations	44
5.2.1 Advantages of using Pulumi's Java APIs	44
5.2.2 Disadvantages of using Pulumi's Java APIs	44
Verbose code	44
5.3 My Scala solution observations	44
5.3.1 Advantages of using my Scala syntactic sugar	44
Very concise code	44
Very readable and elegant code	44
More powerful constructs thanks to the for-comprehension and the monads to act on groups of Outputs	44
5.3.2 Disadvantages of using my Scala syntactic sugar	44
Partial solution, not all the corner cases have been considered .	44
5.4 Final thoughts on the Scala solution	44
Bibliography	47

Elenco delle figure

4.1	Infrastructure Architecture	18
4.2	pulumi up preview	25
4.3	pulumi up confirmed	25
4.4	pulumi destroy preview	26
4.5	pulumi destroy confirmed	26
4.6	Parameters correspondence	31
4.7	vpc methods	33
4.8	Function calls flow and parameters passing	36
4.9	Constructors generation	41

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 Challenges 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

When creating a new Pulumi application, the first step is to define a stack. A stack represents a set of cloud resources that can be managed as a cohesive unit. Each stack is associated with a particular cloud provider (such as AWS, Azure, or GCP), and contains a list of the resources that should be created.

The stack definition typically includes information such as the provider, the region in which the resources should be created, and any required configuration settings. This information is used by Pulumi to provision and manage the resources in the stack.

2.2.2 APIs to define the resources to be created

Once the stack is defined, you write code that implements the resource management logic. Pulumi provides libraries for several programming languages, including TypeScript, Python, Go, .NET, and partially Java. You use these libraries to create and configure cloud resources.

In Pulumi, you define resources using a "resource constructor" function. This function takes in the configuration for the resource as input, and returns a reference to the newly-created resource. You can then use this reference to manage the resource throughout its lifecycle.

Pulumi code can also contain logic for managing relationships between resources. For example, you can specify that one resource depends on another, so that Pulumi knows to create the dependent resource first.

2.2.3 Creating and updating resources with Pulumi commands

Once the code is written, you use Pulumi to create the cloud resources specified in the stack definition. Pulumi uses the provider cloud API to create the resources and corresponding configurations. The resource creation process is declarative, which means that Pulumi automatically figures out the order in which resources should be created based on their dependencies.

Pulumi also supports continuous delivery, meaning that changes to your code are automatically detected and used to update existing resources. When a change is detected, Pulumi compares the desired state (based on your code) to the current state

(based on the resources in the cloud) and makes any necessary changes to bring the resources into compliance with the desired state.

2.2.4 Viewing resources state

Pulumi provides a command to view the state of the resources managed by the application: `pulumi stack`. This command shows the created resources, their properties, and their current state. This information can be used to debug issues and ensure that resources are configured correctly.

The state of a resource is maintained by Pulumi in a "state file". This file contains information about the resources that have been created, as well as their current configuration and state. The state file is automatically updated by Pulumi as resources are created, updated, or deleted.

2.2.5 Restoring resources state

In case of issues or errors during resource creation or management, you can use Pulumi to restore the resource state to the last known good state. This ensures that cloud resources are always consistent with the code and stack definition.

Pulumi uses the state file to track the current state of the resources it manages. If the state of a resource becomes inconsistent with the desired state (for example, if a resource is accidentally deleted), you can use Pulumi to recreate the resource based on the information in the state file. This restores the resource to its last known good state and brings it back into compliance with the stack definition.

2.3 Pulumi's Output

In an IaC context, and therefore with Pulumi, the creation of the resources is not immediate. Hence, in our code when we call a constructor for given resource, we are not actually creating it in that exact moment, but we are rather requesting Pulumi to instantiate such a resource on the given cloud provider. So, the usage of that resource is not available until Pulumi will have completed its creation on the cloud provider. How can we perform operations on a resource when we do not know when it will become available? Pulumi addresses such a problem with the `Output` type.

In Pulumi, `Output` values are typically computed asynchronously, so that they can represent resources that are being provisioned by cloud providers. Like a future, an `Output` can be used to chain operations that depend on the completion of other operations. This feature will come handy to chain the definition of resources that depends on other resources.

We'll see how to create a Monad out of the `Output` type so that we'll be able to boost our syntactic sugar in the Scala version of the case study.

Capitolo 3

Scala, a modern functional and object-oriented programming language

Scala brief overview

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

For the work of the thesis, Scala 3 has been used.

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.

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(people: 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 `people` 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.

Traits

Traits are used to share interfaces and fields between classes. They are similar to Java 8's interfaces. Classes and objects can extend traits, but traits cannot be instantiated and therefore have no parameters.

Here is a simple example of the traits usage:

```
trait Mighty:
  def roar(): Unit

abstract class Animal:
  def name(): Unit

class Lion extends Animal, Mighty:
```

```

    override def name(): Unit = println("Lion")
    override def roar(): Unit = println("The mighty Lion roars!")

class Cat extends Animal:
    override def name(): Unit = println("Cat")

```

The `Lion` class is extending the abstract class `Animal` and also the `Mighty` trait. This is providing the lion the extra `roar()` function.

In Scala is possible to extend from 0 or 1 abstract classes and as many traits we desire.

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 understand how the `for` comprehension concept works let's 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.

Union type

Scala has also the so called "union types". Used on types, the `|` operator creates a so-called union type. The type `A | B` represents values that are either of the type `A` or of the type `B`.

so the function declaration `def foo(a: Int | String) : Unit` is a function that takes as input either an `Int` parameter or a `String` parameter. Now we can write both the following function calls:

```
* foo(5)

* foo("bar")
```

and both will compile (if a valid function body has been provided obviously).

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

The infrastructure we created for the case study of the thesis is an AWS EC2 [VPC](#) hosting 3 private [subnets](#), 3 public subnets, an [internet gateway](#) to let the public subnets connect to the internet and a [routing table](#) to map the public subnets to the internet gateway. The architecture will look like the following:

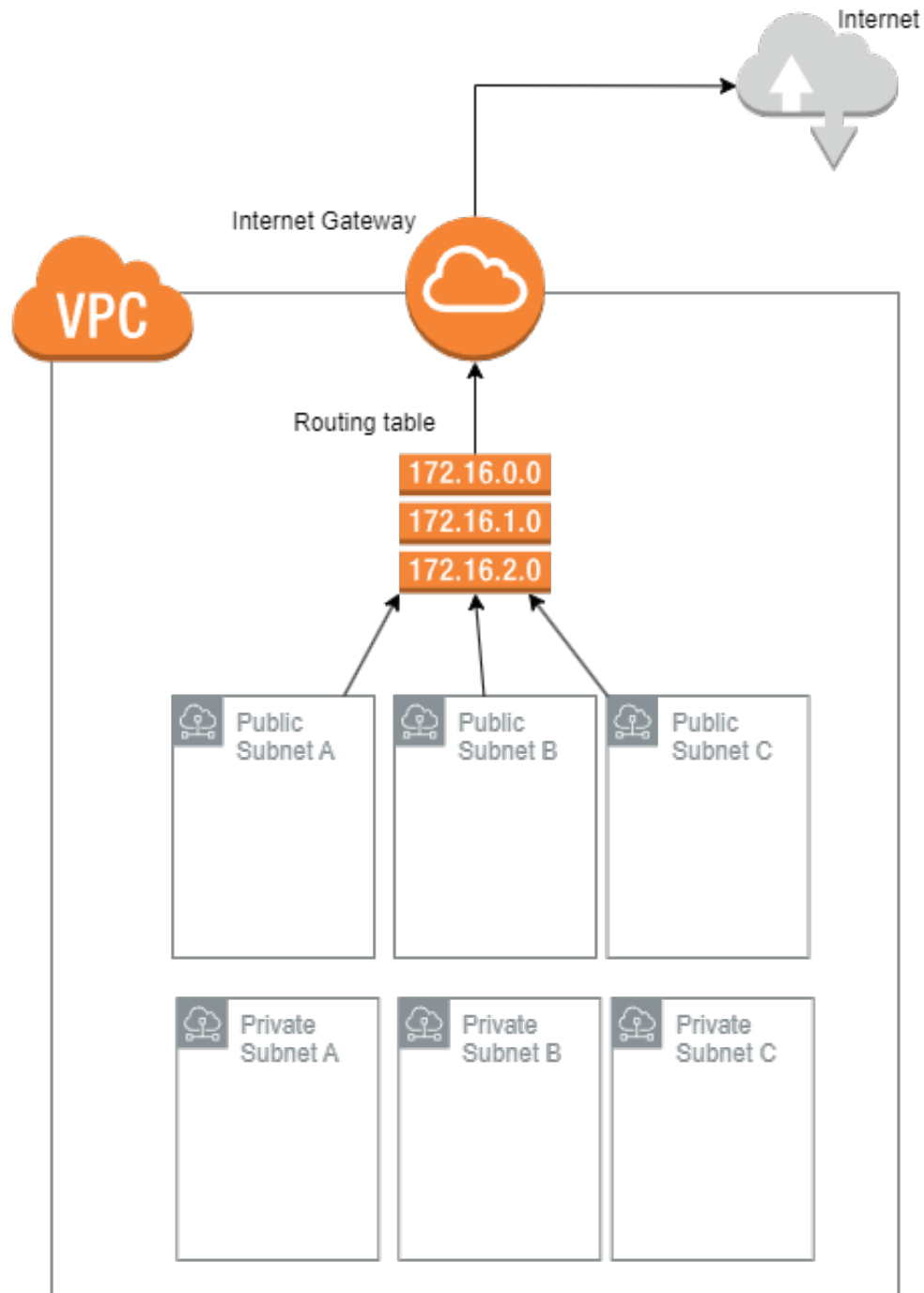


Image 4.1: Infrastructure Architecture

AWS is divided into regions, like us-east-1, eu-central-1, eu-south-1, etc. For the purpose of our thesis eu-south-1 as a region has been chosen. The servers of such a region are located in Milan. Each region can have multiple availability zones, eu-south-1 has 3 different availability zones, and because of this we chose to create 3 couples of public/private subnets (A-A, B-B, C-C). One couple on each availability zone. The purpose of the availability zones is mainly for robustness. If an availability zone becomes temporary unavailable, we can rely on the others to keep our services up.

VPC

The VPC is our "container" for all the other infrastructure resources. We'll define within it the subnets, the internet gateway and the routing table.

The most important setting of our AWS EC2 VPC is the CIDR (Classless Inter-Domain Routing) block. It represents the range of private IP addresses that the VPC can use to create and manage resources within the VPC.

CIDR The CIDR block is used to define the range of IP addresses that the VPC can use. In our case the CIDR block is 10.136.0.0/24, which means that the VPC has access to all IP addresses from 10.136.0.0 to 10.136.0.255. The 24 in the CIDR block is the prefix length, aka the subnet mask, that is used to identify the VPC. The remaining 8 bits of the IP address will be used to identify the hosts in the VPC. We'll assign a name as well to our VPC so that will be easier to recognize it when we'll inspect the AWS Management Console, that is the GUI version of the AWS CLI.

Subnet

The subnets will require the ID of the VPC and the CIDR block that defines their IP scope. For the subnets we'll use the following CIDR blocks:

- * Private Subnet A: 10.136.0.0/27
- * Private Subnet B: 10.136.0.32/27
- * Private Subnet C: 10.136.0.64/27
- * Public Subnet A: 10.136.0.96/27
- * Public Subnet B: 10.136.0.128/27
- * Public Subnet C: 10.136.0.160/27

3 of the 8 bits left out for the hosts identification have been used to identify the subnets. In fact we shall notice that now the subnet mask is not 24 anymore, but 27. Hence, we are left with 5 bits to identify the hosts within each subnet, giving us 32 possible IPs. From such IPs 2 are reserved for the network address and the broadcast address, so we have 30 possible IPs. We won't discuss this topic any further since it isn't essential for the final objective of the thesis.

Moreover, we'll define also the availability zone for each subnet.

InternetGateway

The definition of the internet gateway is actually quite straight forward. The mandatory parameter to assign is the ID of the VPC.

RouteTable

The definition of a route table is required in order to bind the public subnets to the internet gateway, so that they can send and receive data over the internet.

Here, along with the VPC ID, we assign the routes of such routing table. In order to do this we have to provide a CIDR and a target resource to which the packet will be forwarded to, that can be a subnet or the internet gateway in our case. The internet gateway is mapped with the CIDR block 0.0.0.0/0. Obviously also the ID of the internet gateway is required in order to bind it to the routing table.

In a nutshell this means that every packet not directed to a host within the VPC will be routed to the internet gateway and then to the internet.

The association of the public subnets to the routing table will be achieved using the *route table association* resource. Such a resource will require us to provide the id of the subnet and the id of the routing table to establish a connection. The private subnets will not be associated to such a routing table, since we want to keep them private. Differently, without explicitly associating them to a given routing table, AWS will automatically associate them to a default routing table that, being not bound to an internet gateway, will keep them private.

4.3 Typescript implementation of the case study

The first version of the implementation of the previously defined architecture is written using the Typescript APIs of Pulumi. The structure of the project, and this holds for the Java and Scala version as well, is trivial. We have a simple `index.ts` file that defines the entry point for our typescript project, and it is just a couple of lines long:

```
1 import { VPC } from './VPC/VPC';
2
3 const vpc = new MyVPC("Custom VPC");
```

The interesting part relies on the `MyVPC` class. Such a class extends the `ComponentResource` class of Pulumi. In our typescript implementation, we use the constructor of the user-defined `MyVPC` class to call all the class methods that are responsible for the creation of the resources. But this is just mentioned since the interesting part of the code are the actual methods that are responsible for the creation of the resources.

4.3.1 VPC resource creation

The code to the VPC resource creation API of Pulumi is done in this method of the `MyVPC` class:

```
1 protected createVPC() {
2   this.vpc = new Vpc("vpc\_res", {
```

```
3     cidrBlock: "10.136.0.0/24",
4     tags: {
5         Name: "myVPC-typescript",
6     },
7 },
8 {
9     parent: this,
10 });
11 }
```

`vpc_res` is the name that will be given by Pulumi at this resource once on the stack. We can notice how the various parameters are given in a declarative style within the curly brackets. Such a syntax is syntactic sugar for a Map definition. At line 8 the parent of this resource is set. With this specification, we are telling to the stack of Pulumi that the vpc resource `vpc_res` that we are creating is a child resource of the VPC resource identified by the `MyVPC` class.

4.3.2 Internet gateway creation

To create the internet gateway resource we can use the following code:

```
1 protected createIGW(){
2     this.gw = new InternetGateway("gw", {
3         vpcId: this.vpc?.id,
4         tags: {
5             Name: "myIGW-typescript",
6         },
7     },
8     {
9         parent: this.vpc,
10    });
11 }
```

It is really simple since it requires just the ID of the VPC in which it has to be created and optionally a name and a parent for the Pulumi's stack representation.

4.3.3 Subnets creation

To create the private and the public subnets we require a more complex logic. The function that has been used is `protected createAZsSubnets(isPvt: Boolean)`. It is called twice, once with a true value to create the private subnets, and another one with false to instantiate the public ones (and connect them to the routing table bound with the internet gateway).

In the body of the function, first we want to get all the availability zones present in the AWS region we are working on. To achieve this, we will use such a function `this.availableZones = aws.getAvailabilityZonesOutput()`.

Second, we want to create both a private and public subnet in each availability zone acquired with the aforementioned method. The `pulumi.all` function, in combination with the `apply` function will help us in achieving such a goal.

pulumi.all

`pulumi.all` is a utility function in Pulumi that allows you to combine multiple Outputs into a single Output that resolves to an array of the resolved values of each Output. So if we consider the following code:

```
1 pulumi.all([this.availableZones.names, this.vpc!.id])
```

Such a call returns us an `Output<[string[], string]>`. The array of strings is the list of the availability zones names, while the second string represents the ID of our VPC on AWS EC2. Now we have a new Output type that is more suitable to create the subnets based on our VPC ID, because the function `apply` is letting us "open" an `Output` value and access its content.

.apply

Lets extend our code in this way:

```
1 pulumi.all([this.availableZones.names, this.vpc!.id]).apply(([
    azNames, vpcId]) => {
2     // lambda's body to create the subnets here
3 })
```

The `apply` function is letting us access `Output<[string[], string]>` and apply some logic on the inner values.

The apply's lambda Now that we have the access to the list of availability zones and the VPC ID, we can iterate over the availability zones and create the subnets for our VPC. Here is the complete code of the function:

```
1 protected createAZsSubnets(isPvt: Boolean){
2     this.availableZones = aws.getAvailabilityZonesOutput()
3     pulumi.all([this.availableZones.names, this.vpc!.id]).apply(
4         ([azNames, vpcId]) => {
5             let i = 0
6             let listToPushInto: Subnet[] = Array<aws.ec2.Subnet>()
7             azNames.forEach(azName => {
8                 let compName = azName + (isPvt ? "-pvt" : "-pub") + "-
9                     subnet-typescript"
10                listToPushInto.push(new Subnet(compName, {
11                    vpcId: vpcId,
12                    availabilityZone: azName,
13                    cidrBlock: isPvt ? this.pvtSubnetsCidrs[i] : this.
14                        pubSubnetsCidrs[i],
15                    tags: {
16                        Name: compName,
17                    },
18                },
19                {
20                    parent: this.vpc
```

```

18     }
19     ));
20     i++;
21   });
22   if(!isPvt){
23     this.pubSubNets = listToPushInto
24     // attaching the route table to the pub sub nets
25     this.attachRouteTableToPubSubnets()
26   }
27   else
28     this.prvSubNets = listToPushInto
29   });
30 }

```

In the code we instantiate private or public subnets basing on the `isPvt` passed in the `createAZsSubnets`. We can notice that for the creation of a subnet we pass arguments such as `vpcId`, the availability zone name, the CIDR block, and a name (that is a tag) to better identify it on the stack.

Each newly created subnet is then pushed into the class field `this.prvSubNets` or `this.pubSubNets` (that are arrays of subnets), basing on the nature of the subnet. The `this.pubSubNets` will be used by the `this.attachRouteTableToPubSubnets()` function to bind the public subnets to the internet gateway through the routing table. The crucial point here is that in order to bind the public subnets to the routing table, we must call `this.attachRouteTableToPubSubnets()` as the last step in our `pulumi.all` function call. This ensures that we have all the necessary subnets to make the binding, since the call will be done at the completion of the creation of the subnets.

4.3.4 Routing table creation

This is the code to create the routing table resource:

```

1  protected createRouteTable() {
2    this.routeTable = new RouteTable("example", {
3      vpcId: this.vpc!.id,
4      routes: [
5        {
6          cidrBlock: "0.0.0.0/0",
7          gatewayId: this.gw!.id,
8        },
9      ],
10     tags: {
11       Name: "myRouteTable-typescript",
12     },
13   },
14   {
15     parent: this.vpc,
16   });
17 }

```

On top of the classic VPC ID we are assigning here the routes. As we mentioned before in the [RouteTable](#) paragraph, we are defining the route with CIDR 0.0.0.0/0 to

redirect all the packets coming from the public subnets, and not having as destination an IP internal at our VPC, to the internet gateway.

We are also giving a name to the routing table and assigning its parent.

4.3.5 Attaching the public subnets to the internet gateway

As we mentioned previously, we use the `this.attachRouteTableToPubSubnets()` function to attach the public subnets to the internet gateway. Here is the code of the function:

```

1 protected attachRouteTableToPubSubnets(){
2   let i = 0
3   this.pubSubNets.forEach(subNet => {
4     new aws.ec2.RouteTableAssociation(`${i}-
      routeTableAssociation-typescript`, {
5       subnetId: subNet.id,
6       routeTableId: this.routeTable!.id,
7     },
8     {
9       parent: this.vpc
10    });
11    i++
12  });
13 }
```

The code is quite straight forward. We are defining a new `RouteTableAssociation` AWS EC2 resource that requires just the id of the subnet and the id of the routing table to which we want to attach the subnet to.

4.4 Creating the resources with Pulumi

After having seen all the code to create the resources, we'll see what the Pulumi command `pulumi up` will do. The command checks if all the resources that we want to create have valid parameters and there are not circular dependencies among the resources on their creation. If everything is nice and neat, it will shows us the preview of the changes that we are about to get:


```

Previewing update (dev)

View Live: https://app.pulumi.com/Cis8/scala/dev/previews/c7370a40-7d38-41b7-820f-1329afafe1f2

  Type                                         Name                                         Plan
+ pulumi:pulumi:Stack                         scala-dev                                    create
+   └─ VPC                                    My Custom TS VPC                           create
+       └─ aws:ec2:Vpc                        my-vpc-main                                create
+           └─ aws:ec2:InternetGateway        gw                                           create
+               └─ aws:ec2:RouteTable          myRouteTable                               create
+                   └─ aws:ec2:Subnet          eu-south-1a-pub-subnet-typescript          create
+                       └─ aws:ec2:Subnet      eu-south-1a-pvt-subnet-typescript          create
+                           └─ aws:ec2:Subnet  eu-south-1b-pvt-subnet-typescript          create
+                               └─ aws:ec2:Subnet  eu-south-1b-pub-subnet-typescript          create
+                                   └─ aws:ec2:Subnet  eu-south-1c-pvt-subnet-typescript          create
+                                       └─ aws:ec2:Subnet  eu-south-1c-pub-subnet-typescript          create
+                                           └─ aws:ec2:RouteTableAssociation  0-assoc-typescript                        create
+                                               └─ aws:ec2:RouteTableAssociation  1-assoc-typescript                        create
+                                                   └─ aws:ec2:RouteTableAssociation  2-assoc-typescript                        create

Resources:
  + 14 to create

Do you want to perform this update? [Use arrows to move, type to filter]
[experimental] yes, using Update Plans (https://pulumi.com/updateplans)
> yes
  no
  details

```

Image 4.2: pulumi up preview

If we press yes this is the output:

```

Do you want to perform this update? yes
Updating (dev)

View Live: https://app.pulumi.com/Cis8/scala/dev/updates/28

  Type                                         Name                                         Status
+ pulumi:pulumi:Stack                         scala-dev                                    created (0.94s)
+   └─ VPC                                    My Custom TS VPC                           created
+       └─ aws:ec2:Vpc                        my-vpc-main                                created (1s)
+           └─ aws:ec2:InternetGateway        gw                                           created (0.64s)
+               └─ aws:ec2:Subnet          eu-south-1a-pvt-subnet-typescript          created (0.94s)
+                   └─ aws:ec2:Subnet      eu-south-1a-pub-subnet-typescript          created (1s)
+                       └─ aws:ec2:Subnet  eu-south-1b-pvt-subnet-typescript          created (1s)
+                           └─ aws:ec2:Subnet  eu-south-1b-pub-subnet-typescript          created (1s)
+                               └─ aws:ec2:Subnet  eu-south-1c-pvt-subnet-typescript          created (2s)
+                                   └─ aws:ec2:Subnet  eu-south-1c-pub-subnet-typescript          created (2s)
+                                       └─ aws:ec2:RouteTable          myRouteTable                               created (2s)
+                                           └─ aws:ec2:RouteTableAssociation  0-assoc-typescript                        created (0.57s)
+                                               └─ aws:ec2:RouteTableAssociation  1-assoc-typescript                        created (0.86s)
+                                                   └─ aws:ec2:RouteTableAssociation  2-assoc-typescript                        created (1s)

Resources:
  + 14 created

Duration: 22s

```

Image 4.3: pulumi up confirmed

We can notice how the resources created are nested into each other thanks to the parent option that we used. This is helping us in keeping our resources on the stack

nicely ordered and tied up.

4.5 Destroying the resources with Pulumi

Now let's use the `pulumi destroy` command to destroy the resources on our Pulumi's stack. The preview of the changes that we are about to get look like this:

```

Previewing destroy (dev)

View Live: https://app.pulumi.com/Cis8/scala/dev/previews/b9e0652e-9143-4a31-b9cc-f24cbc752bf4

  Type                                         Name                                         Plan
- pulumi:pulumi:Stack                         scala-dev                                    delete
-   └─ VPC                                    My Custom TS VPC                           delete
-     └─ aws:ec2:Vpc                           my-vpc-main                                delete
-       ├── aws:ec2:RouteTableAssociation      2-assoc-typescript                         delete
-       ├── aws:ec2:RouteTableAssociation      0-assoc-typescript                         delete
-       ├── aws:ec2:RouteTableAssociation      1-assoc-typescript                         delete
-       ├── aws:ec2:RouteTable                 myRouteTable                              delete
-       ├── aws:ec2:Subnet                     eu-south-1b-pvt-subnet-typescript          delete
-       ├── aws:ec2:Subnet                     eu-south-1a-pub-subnet-typescript          delete
-       ├── aws:ec2:Subnet                     eu-south-1b-pub-subnet-typescript          delete
-       ├── aws:ec2:InternetGateway            gw                                           delete
-       ├── aws:ec2:Subnet                     eu-south-1c-pub-subnet-typescript          delete
-       ├── aws:ec2:Subnet                     eu-south-1c-pvt-subnet-typescript          delete
-       └─ aws:ec2:Subnet                     eu-south-1a-pvt-subnet-typescript          delete

Resources:
- 14 to delete

Do you want to perform this destroy? [Use arrows to move, type to filter]
> yes
  no
  details
  
```

Image 4.4: pulumi destroy preview

If we confirm the changes this is the result:

```

Do you want to perform this destroy? yes
Destroying (dev)

View Live: https://app.pulumi.com/Cis8/scala/dev/updates/29

  Type                                         Name                                         Status
- pulumi:pulumi:Stack                         scala-dev                                    deleted
-   └─ VPC                                    My Custom TS VPC                           deleted
-     └─ aws:ec2:Vpc                           my-vpc-main                                deleted (0.91s)
-       ├── aws:ec2:RouteTableAssociation      2-assoc-typescript                         deleted (0.55s)
-       ├── aws:ec2:RouteTableAssociation      0-assoc-typescript                         deleted (0.73s)
-       ├── aws:ec2:RouteTableAssociation      1-assoc-typescript                         deleted (1s)
-       ├── aws:ec2:RouteTable                 myRouteTable                              deleted (0.68s)
-       ├── aws:ec2:InternetGateway            gw                                           deleted (0.62s)
-       ├── aws:ec2:Subnet                     eu-south-1c-pub-subnet-typescript          deleted (0.98s)
-       ├── aws:ec2:Subnet                     eu-south-1b-pub-subnet-typescript          deleted (1s)
-       ├── aws:ec2:Subnet                     eu-south-1a-pvt-subnet-typescript          deleted (1s)
-       ├── aws:ec2:Subnet                     eu-south-1b-pvt-subnet-typescript          deleted (1s)
-       ├── aws:ec2:Subnet                     eu-south-1a-pub-subnet-typescript          deleted (2s)
-       └─ aws:ec2:Subnet                     eu-south-1c-pvt-subnet-typescript          deleted (2s)

Resources:
- 14 deleted

Duration: 10s

The resources in the stack have been deleted, but the history and configuration associated with the stack are still maintained.
If you want to remove the stack completely, run `pulumi stack rm dev`.
  
```

Image 4.5: pulumi destroy confirmed

4.6 My Scala implementation of the case study

Since Scala is not supported by Pulumi, I had to implement it on my own. The Pulumi team is clear on the official way to add the support of a new language for Pulumi, and the procedure is long and laborious, too laborious for a master thesis. The full procedure can be found on [New Language Bring Up](#).

The idea behind the adopted solution is to exploit the compatibility of Scala with the Java libraries to write custom syntactic sugar. Such syntactic sugar will be based on the Pulumi Java's APIs and will provide to the user cool constructs to write readable and expressive code to interact with Pulumi.

The steps of the work done have been the followings:

1. manually write the *sugarized* functions to create the Pulumi resources using Scala
2. use such functions to recreate the Stack obtained with the typescript solution shown before in the [Typescript implementation of the case study](#) section
3. create an automatic code generator for our syntactic sugar functions, so that we can quickly create a library for Scala's Pulumi APIs
4. try to recreate the stack with the automatically generated code

Obviously, the third step is quite wide, and in fact with my work I had the time to generate only the functions for a part of the Java's Pulumi APIs for the AWS EC2 module.

Now the just defined steps will more accurately be presented.

4.6.1 Structure of the Java APIs for the constructors of the resources in Pulumi

To understand the syntactic sugar functions that I defined, let's first consider the general structure of the Java APIs for the constructors of the resources.

The constructor of a resource, in general accepts a name and an instance of the corresponding `Args` class of the resource we are creating. Let's consider for example the `Vpc` resource. In Java, to instantiate such a resource we'd call:

```
1 protected Vpc vpc = new Vpc("my-vpc-java", VpcArgs.builder()  
2     .cidrBlock("10.136.0.0/24")  
3     .tags(Map.of("Name", "main"))  
4     .build(),  
5         CustomResourceOptions.builder()  
6             .parent(this)  
7             .build());
```

We can (hardly) see that along with the name to be assigned to the vpc "my-vpc-java", a `VpcArgs` builder and a `CustomResourceOptions` builder are passed by. These builders

will create an instance of the respective classes that will be used to set respectively the parameters and the parent of the Vpc resource. So, for our case study we need to consider: the name to be assigned at the created resource on the Pulumi stack, the builder of the respective `Args` class of the resource, and the `CustomResourceOptions` builder.

4.6.2 Syntactic sugar usage

Our syntactic sugar is split in 2 categories of functions. The first is about the functions that represent the constructors of the resources. The second is for the methods available within the builders of the `Args` classes and for the `CustomResourceOptions` builder functions. The idea to create a resource is to call the *sugarized* function that represent the constructor of that resource, and then call the Builder methods to assign the various parameters to the resource.

Vpc creation

This is how a Vpc resource can be created with my syntactic sugar:

```
1 val myVpc: Vpc = vpc("scala-main") ({
2   cidrBlock("10.136.0.0/24")
3   tags("Name" -> "myVpcScala")
4 },{
5   parent(this)
6 })
```

At line 1, the `vpc` function is the actual *sugarized* function for the VPC resource constructor. In fact we can notice that we have a curried function. The first parentheses is taking the parameter for the resource name on the Pulumi stack, while the second one is containing two lambdas (defined by the curly brackets). These lambdas are respectively used to call all the builder methods of the `VpcArgs` class and the ones for the `CustomResourceOptions`.

We can notice that we didn't explicitly defined an instance of the builders of such classes. We will soon see how we achieved such a syntactic sugar trick.

The `cidrBlock` and the `tags` methods are the generated methods for builder of the `VpcArgs` class.

The `parent` method instead is the generated method for the builder of the `CustomResourceOptions` class.

Moreover, we can also notice that inside tags, that expects a `Map[String, String]` type, the instantiation of a `Map[String, String]` containing a single elements isn't required. This other trick will be explained later as well.

Internet gateway creation

Much similar to the VPC resource, we have this code for the internet gateway creation:

```
1 val myIGW: InternetGateway = internetGateway("gw") ({
2   vpcId(myVpc.getId())
3   tags("Name" -> "myIGWScala")
4 },{
5   parent(myVpc)
```

```
6 })
```

The code won't be commented since is analogous to the VPC case.

Routing table creation

The code to create a routing table:

```
1 val myRouteTable = routeTable("myRouteTable") ({
2   vpcId(myVpc.getId())
3   routes(
4     routeTableRouteArgs({
5       cidrBlock("0.0.0.0/0")
6       gatewayId(myIGW.getId())
7     })
8   tags("Name" -> "myRouteTableScala")
9 }, {
10  parent(myVpc)
11 })
```

The only thing that is worth to mention here is that the `routes` function, at line 3, expects a `List[RouteTableRouteArgs]`, but we are providing only a `RouteTableRouteArgs`. As for the case of the `Map[String, String]` with the `parent` method mentioned above, the same trick has been used to provide syntactic sugar that lifts us from the need of instantiate a singleton `List[RouteTableRouteArgs]` manually.

Subnets creation

Much different from the other resources is the function to create the subnets:

```
1 def createAzSubnets(isPvt: Boolean) =
2   availabilityZonesNames().map((az: GetAvailabilityZonesResult) =>
3     for
4       (name, cidr) <- az.names().zip(if isPvt then pvtSubnetsCidrs else
5         pubSubnetsCidrs)
6     yield
7       val compName = name + "-" + (if isPvt then "pvt" else "pub") + "-
8         subnet-scala"
9       subnet(compName) ({
10        vpcId(myVpc.getId())
11        availabilityZone(name)
12        cidrBlock(cidr)
13        tags("Name" -> compName)
14      }, {
15        parent(myVpc)
16      })
17   )
```

We can notice here how we had the possibility to use the `map` function on `availabilityZonesNames()`. `availabilityZonesNames()` returns a `Output[GetAvailabilityZonesResult]` value, that isn't directly accessible from the `map` function. To achieve such a result we

made a monad out of the `Output` type, so that `map` can be applied on the inner value of the `Output` context. We'll see better how is the actual implementation of the monad soon.

The `for yield` construct here is iterating over a zipped list made out of the availability zones names and the CIDR blocks for the respective kind of subnet to create (private or public, based on the input parameter `isPvt`). The `for yield`, for each tuple `(name, cidr)` will create a `Subnet` value and will return a `Subnet[]` as output. Now, the `for` is still inside the lambda of the `availabilityZonesNames().map` function. Consider the fact that `map` takes an `Output[A]` and returns an `Output[B]`, where `A = GetAvailabilityZonesResult` and `B = Output[Iterable[Subnet]]`. This is different from the typescript implementation in the [Apply's lambda](#) paragraph, where the function that creates the subnets returns a plain `Subnet[]`.

We will make further considerations about such a difference in the [Comparison between the languages for Pulumi and the advantages of Scala](#) chapter.

Attaching the subnets to the routing table

```
1 def attachRouteTableToPubSubnets(): Output[Iterable[
    RouteTableAssociation]] =
2   pubSubnets.map((subnets: Iterable[Subnet]) =>
3     for
4       (ps, idx) <- subnets.zipWithIndex
5     yield
6       routeTableAssociation(idx + "-assoc-scala") ({
7         subnetId(ps.getId())
8         routeTableId(myRouteTable.getId())
9       }, {
10        parent(myVpc)
11      })
12   )
```

Similarly to the subnet creation function, also here we use `map` to "unbox" an `Output[Iterable[Subnet]]` value, to apply some logic on the inner value and then return a `Output[Iterable[RouteTableAssociation]]` value as result. Since the logic is analogous we won't comment this code further, but the fact we're having these two functions that both take an `Output[A]` as input and return an `Output[B]` will be a key point for our observations in the [Comparison between the languages for Pulumi and the advantages of Scala](#) chapter.

4.6.3 Syntactic sugar for the constructors of the resources

All the methods that we just used for creating the Pulumi resources in Scala (`vpc`, `internetGateway`, ecc.), behind the scenes are implemented following a common pattern. Consider the `vpc` function of the syntactic sugar defined inside the "PulumiUtilFunctionsForScala.scala" file:

```
1 def vpc(param: String)
2   (init: VpcArgs.Builder ?=> Unit,
3     initOpt: (CustomResourceOptions.Builder ?=> Unit) =
4       baseOpts): Vpc =
```

```

4      given b: VpcArgs.Builder= VpcArgs.builder()
5      init
6      given bo: CustomResourceOptions.Builder =
          CustomResourceOptions.builder()
7      initOpt
8      new Vpc(param, b.build(), bo.build())

```

Lets analyze the function by steps. First of all we can see that the function declaration is curried, we have 2 parentheses with different input parameters.

The first parentheses take simply a string parameter, that is used to set the name of the `Vpc` resource on the Pulumi stack.

The second parentheses are taking two lambdas as parameters: a `VpcArgs.Builder ?=> Unit` and a `CustomResourceOptions.Builder ?=> Unit`, with default parameter `baseOpts`, that we'll see in a moment what is.

In Scala, a lambda that takes an `Int` and returns a `String` has this type notation: `Int => String`, so does that '?' in front of the '=' mean?

Such '`?=>`' is denoting a context function, that is a function with (only) context parameters. In the [Given and using keywords](#) paragraph we introduced the `using` keyword. Such '?' is quite analogous to a `using` keyword used to mark an function's input parameter as a context parameter. This is, in part, what let us call the builder methods like `cidrBlock("10.136.0.0/24")` and `tags("Name" -> "myVpcScala")` (as shown in the code shown in the [Vpc creation in Scala](#) paragraph) without having to call them on a specific builder instance. We will get the whole picture of this trick when we'll talk about the syntactic sugar for the builder methods in the [Syntactic sugar for the builders' methods](#) paragraph.

Correspondence between the input parameters and the user defined code used to create the VPC

To have a better idea to what these parameters refer in our VCP creation case, consider this image:

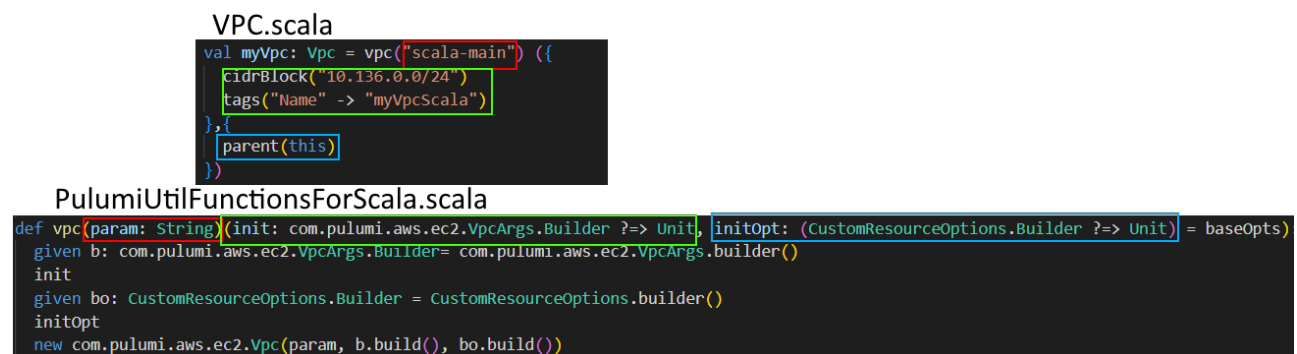


Image 4.6: Parameters correspondence

The red box represents the name for our VPC resource on the Pulumi stack.

The green one, with the curly brackets, is the lambda that takes a *given* `VpcArgs.Builder`

as an implicit parameter from the context. In fact we are not providing any explicitly. The `given b` defined in the first line of the body of the `vpc` function is the instantiation of a *given* instance of such a builder, that will be automatically injected by the compiler in the `init` lambda represented by the green box. Analogous is the concept for the blue box.

Vpc construction

Now that we understood what are the input parameters of our `vpc` function and to what do they correspond in the resource creation shown in the [Syntactic sugar usage](#) section, we can see how the actual creation of the resource is made. After having executed the `init` and `initOpt` lambdas, that behind the scenes will set the parameters of the respective *given* builders, we can create the resource using the constructor offered by the Pulumi Java APIs. `new Vpc(param, b.build(), bo.build())` is a direct call to such libraries, and will create actually create the VPC.

The baseOpts function

The `baseOpts` function that we mentioned before as a default lambda for our `vpc` function is the following:

```
1 def baseOpts(using o: CustomResourceOptions.Builder) : Unit =
    {}
```

In practice, is an vacuous lambda that does nothing on the `CustomResourceOptions` builder. The question here is: *why do we need such a default function?* To answer the question let's consider one more time the code to create a VPC with out syntactic sugar:

```
1 val myVpc: Vpc = vpc("scala-main") ({
2   cidrBlock("10.136.0.0/24")
3   tags("Name" -> "myVpcScala")
4 },{
5   parent(this)
6 })
```

We can see that we passed both the lambdas for the `VpcArgs` builder and for the `CustomResourceOptions` builder, but what if we want to simply use the `VpcArgs` builder and not set the parent? We can do the following:

```
1 val myVpc: Vpc = vpc("scala-main") {
2   cidrBlock("10.136.0.0/24")
3   tags("Name" -> "myVpcScala")
4 }
```

This code compiles, and we can notice that we even got rid of the parentheses around the two groups of curly brackets for the lambdas. If this compiles is only thanks to the default parameter that is automatically injected in by the compiler, as we didn't give

an explicit one.

Another question now might arise: *why didn't we change the signature of the function in the following way?*

```
1 def vpc(param: String)(using initOpt: (CustomResourceOptions.  
    Builder ?=> Unit), init: VpcArgs.Builder ?=> Unit) : ecc.
```

Here we have set the `initOpt` as an implicit parameter with the `using` keyword. The main problems that we face with this solution are mainly 2: we have to swap the order of the parameters and we will be forced to use the `using` keyword to explicitly pass a custom lambda for the `CustomResourceOptions` builder.

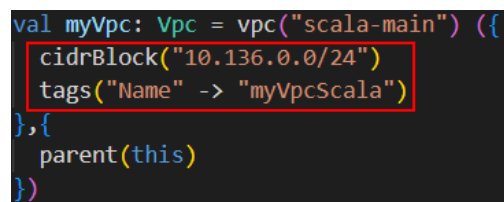
The first problem leads to a sort of awkwardness while defining the VPC resource, since we have to define first the parent and then the actual parameters of the VPC resource.

The second problem is requiring us to write `using` every time we want to pass an explicit and custom lambda that sets some parameters of the `CustomResourceOptions` builder, that is annoying.

These problems are due to the functioning of the Scala language. An implicit parameter must come before all the explicit parameters and when trying to use an explicit parameter in place of an implicit one we must use the `using` keyword. So, the original solution with the default parameter is the best one since it doesn't require us to swap the order of the parameters and we are totally free to choose whether to pass or not the explicit lambda for the `CustomResourceOptions` builder without worsening our syntactic sugar.

4.6.4 Syntactic sugar for the builders' methods

What we have shown up to now is not enough to have our syntactic sugar working, we are missing a subtle point to get the work done. Lets pay attention to how the `VpcArgs.Builder` parameters are set inside the `vpc` function call. To be precise we are referring to these methods:



```
val myVpc: Vpc = vpc("scala-main") ({  
    cidrBlock("10.136.0.0/24")  
    tags("Name" -> "myVpcScala")  
}, {  
    parent(this)  
})
```

Image 4.7: vpc methods

Once such a lambda will be invoked inside the `vpc` function itself, these methods will be executed, but on which builder?

We already mentioned the fact that inside the `vpc` function a context instance of the `VpcArgs.Builder` is initialized and the `init` lambda is able to use it as input parameter thanks to the `'?=>'` operator we have seen. But how can the actual `cidrBlock` and `tags` methods know on which builder they are being invoked?

Without surprise, those methods take the `VpcArgs.Builder` builder as an implicit parameter with the `using` keyword.

This is the signature for the `cidrBlock` method in the syntactic sugar file named `"PulumiBuilderUtilFunctionsForScala.scala"`:

```
1 def cidrBlock(param: String | Output[String]) (using b: cidrBlockOwners)
  : Unit
```

We can notice that we have once more a curried function. Anyway, from how we have seen before, the `cidrBlock` (and analogously for all the other builder methods) is called with just a single set of parentheses. This is due to the fact that the second parentheses here are taking an implicit parameter, properly marked with the `using` keyword.

The `param` parameter is, as we have seen in the [Union type](#) paragraph, a union type. The `String | Output[String]` type is defined so since the Pulumi Java APIs for the builders' methods accept both a `String` and an `Output[String]`. Actually, in the Java implementation an overloading of methods is given since the union type of Scala is not available.

The second parentheses are taking an implicit parameter `b` of the type `cidrBlockOwners`, that is defined as follows:

```
1 type cidrBlockOwners = RouteTableRouteArgs.Builder | SubnetArgs.Builder
  | VpcArgs.Builder
```

This is a user defined type that I defined to match the builders of all the `Args` classes that are interested in having such a parameter to assign on their builder instance. In fact in the Java APIs of Pulumi we have many different `Args` classes' builders that want to assign the same parameter (aka. `cidrBlock`) to their own builder. I remind that in the AWS EC2 module there are much more builders of the `Args` classes that define a `cidrBlock` method, but my syntactic sugar has created the methods for only the classes that I used in the case study. This choice has been made also for simplicity in presenting the work done, otherwise the `cidrBlockOwners` type would have been featuring many tens of types. The fact we used a union type to define this function has two main motivations. The first is a Scala language constraint that we came across. Let's say that we wanted to define a function to assign the CIDR block working exclusively for `VpcArgs.Builder` class. A definition of such a function would look like:

```
1 def cidrBlock(param: String | Output[String]) (using b: VpcArgs.Builder)
  : Unit =
2   param match
3     case x: String => builder.cidrBlock(x)
4     case x: Output[String] => builder.cidrBlock(x)
```

And now let's define another function that is working for the `RouteTableRouteArgs.Builder`:

```
1 def cidrBlock(param: String | Output[String]) (using b:
  RouteTableRouteArgs.Builder): Unit =
2   param match
3     case x: String => builder.cidrBlock(x)
4     case x: Output[String] => builder.cidrBlock(x)
```

First, we can notice that they are actually the same, except for the signature, but such a

solution is not going to compile if we try to call the method `cidrBlock("10.136.0.0/24")` here:

```
1 val myVpc: Vpc = vpc("scala-main") ({
2   cidrBlock("10.136.0.0/24") \\ ERROR
3   tags("Name" -> "myVpcScala")
4 },{
5   parent(this)
6 })
```

The compiler will tell us that an ambiguous function call is present at the line 2 of this block of code. This is due to the fact that the functions we defined are curried and their type is `(String | Output[String]) => (VpcArgs.Builder => Unit)` and `(String | Output[String]) => (RouteTableRouteArgs.Builder => Unit)` respectively. When we call `cidrBlock("10.136.0.0/24")` on the line 2 of the code showed above, we are partially applying the curried function and so the compiler doesn't know which function we are trying to call, since it can't infer the exact function call basing only on a different return type (that is the only difference in the two functions).

The second reason is that our *all-in-one* solution is reducing the size of the generated *sugarized* code, since we have just one single method instead of having as many as the builders of the `Args` classes that require that methods are.

Now we are ready to present the entire `cidrBlock` method:

```
1 def cidrBlock(param: String | Output[String]) (using b: cidrBlockOwners)
2   : Unit =
3   b match
4     case builder: RouteTableRouteArgs.Builder =>
5       param match
6         case x: String => builder.cidrBlock(x)
7         case x: Output[String] => builder.cidrBlock(x)
8     case builder: SubnetArgs.Builder =>
9       param match
10        case x: String => builder.cidrBlock(x)
11        case x: Output[String] => builder.cidrBlock(x)
12    case builder: VpcArgs.Builder =>
13      param match
14        case x: String => builder.cidrBlock(x)
15        case x: Output[String] => builder.cidrBlock(x)
```

The body of the function is quite simple in its functioning. It uses the pattern matching to match the correct builder type and then uses pattern matching once more to match the `param` parameter to a `String` or an `Output[String]`. Finally it calls the Java API of Pulumi to set the `cidrBlock` parameter on the builder instance `b`.

The fact we have duplicated code here is inevitable. This is the only solution since if we try to split the duplicated code into an helper function, we would fall again in the ambiguous call error presented above. But since this is automatically generated code, it is not a real problem to have some duplicated code.

To have the final picture of all the functioning lets consider this image:

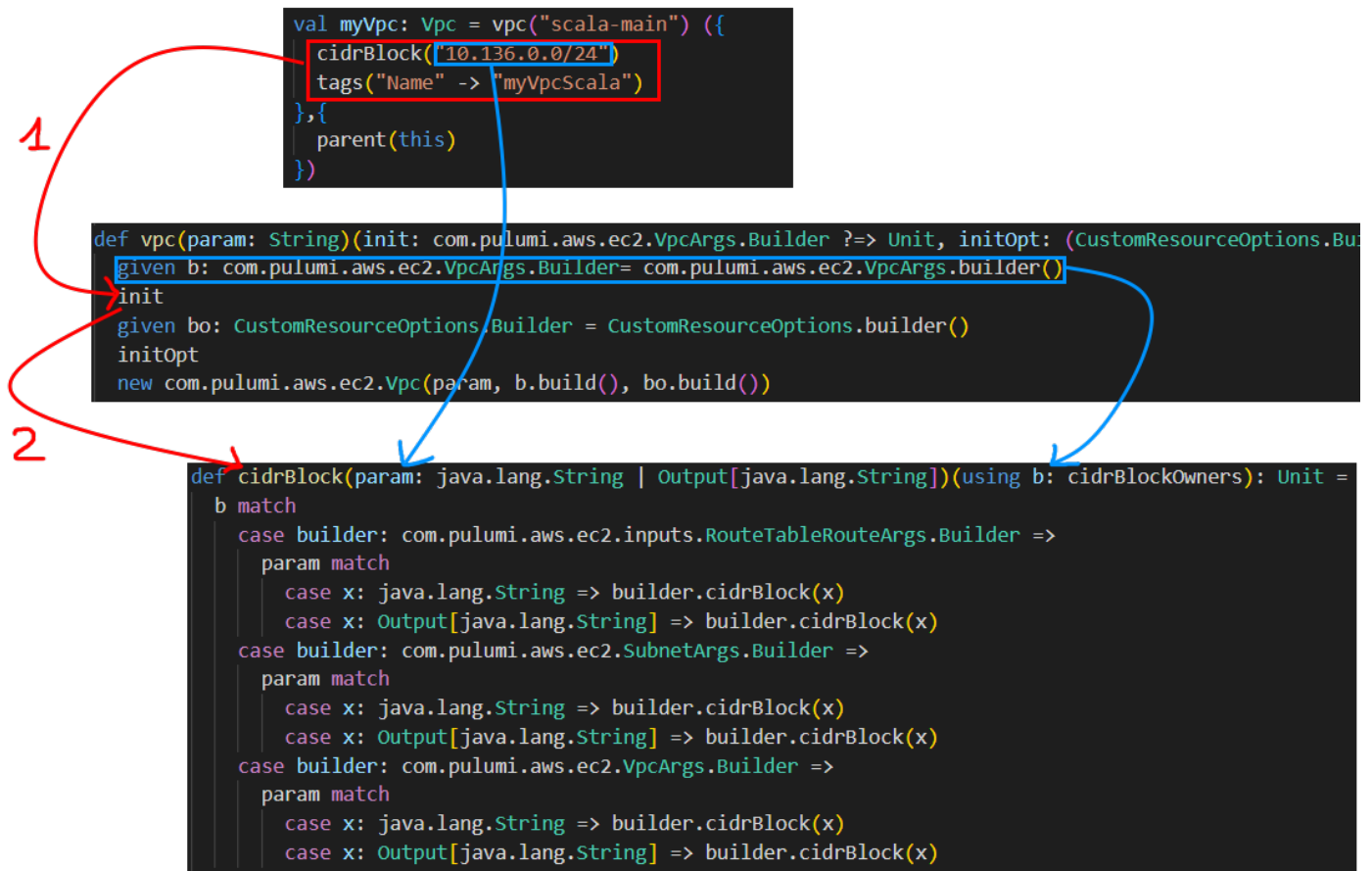


Image 4.8: Function calls flow and parameters passing

We can see how the lambda with the calls to our defined methods `cidrBlock` and `tags` is passed as the `init` parameter of the *sugarized* `vpc` function. Inside the `vpc` function we execute that lambda and so, the `cidrBlock` method is invoked.

In blue we can see from where the parameters of the `cidrBlock` method are coming from. The `String` representing the CIDR block is coming directly from the explicit parameter that we passed, while the `VpcArgs` builder is implicitly injected from the compiler since a *given* instance is defined inside the `vpc` function and the `b` parameter of the `cidrBlock` method is marked with `using`.

Implicit conversion functions

On top of this, to boost our syntactic sugar I defined also two extra functions: `tupleToMap` and `elemToList`. The purpose of these functions is to achieve the tricks that I mentioned previously (in the [Vpc creation in Scala](#) and in the [Routetable creation in Scala](#)) about not needing to explicitly instantiate a singleton Map and a singleton List while passing a single argument to the `tags` or the `routes` methods. The `tupleToMap` function is implemented like this:

```

1 given tupleToMap[A, B]: Conversion[(A, B), Map[A, B]] =
2   (tuple: (A, B)) => Map(tuple)

```

We can notice that the function that converts our tuple into a singleton Map is based on the `Conversion` class of Scala. When a suitable argument for the conversion of type `(A, B)` is found in the code, and a `Map[A, B]` type is expected, then the compiler will apply the conversion to that type. This is exactly what happens with our tuple as single parameter passed to the `tags` method.

And the `elemToList` function is defined in this way instead:

```

1 given elemToList[A <: ResourceArgs]: Conversion[A, List[A]] =
2   (elem: A) => List(elem)

```

The functioning is analogous to `tupleToMap`, but here we added the extra constraint that `A` must be a subtype of the `ResourceArgs` type. This will prevent undesired too generic conversions that could create problems in the compilation of our program.

4.6.5 Functor and Monad implementation for the Output type

After having introduced the concept of functor and monad in the [Functor and monads](#) section, it is here show how the implementation of the monad for the `Output` type has been achieved.

Since a monad is also a functor, let's see how `Functor[Output]` has been implemented.

Functor implementation

First a `Functor` trait has been defined:

```

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

```

A type `Output` to be a functor has to implement a `map` method, that as we have already seen is provided as an extension method.

The functor for the `Output` type is implemented like this:

```

1 given Functor[Output] with
2   extension [A](oa: Output[A])
3     def map[B](f: A => B): Output[B] =
4       oa.applyValue(f.asJava)

```

The implementation of the `map` method relies on the Java APIs of Pulumi, where the `applyValue` method from the `Output` class is provided.

The signature of the given method is `default <U> Output<U> applyValue(Function<T, U> func)`. We are interested in observing that this method takes a function that transforms a value of a type `T` in a value of type `U`, and then it returns an `Output[U]` value as result. This signature is exactly the one we need.

In fact it is sufficient to pass the function `f` taken in input from the `map` invocation and

pass it directly to the `applyValue` method. To be precise, being `f` a Scala function, we need to convert it to a Java function before passing it to `applyValue`. We achieve this by using the `.asJava` method from the `scala.jdk.FunctionConverters._` conversion library for Scala.

Monad implementation

As for the `Functor`, the `Monad` trait has been defined:

```
1 trait Monad[F[_]] extends Functor[F]:
2   // The unit value for a monad
3   def pure[A](x: A): F[A]
4
5   extension [A](x: F[A])
6     // The fundamental composition operation
7     def flatMap[B](f: A => F[B]): F[B]
8     // The 'map' operation can now be defined in terms of 'flatMap'
9     def map[B](f: A => B) = x.flatMap(f.andThen(pure))
```

A monad, to be called so, must define a `pure` function. I remind that such a method puts a value inside a context.

Then the `flatMap`, that is the other method required from a monad, is defined as an extension method.

The `map` method instead can be now be redefined using `flatMap`. This is letting us not depend any more on the `.applyValue` from the Pulumi libraries for Java, because we can now use `flatMap` to achieve the desired result.

The monad for the `Output` type is implemented in this way:

```
1 given Monad[Output] with
2   def pure[A](x: A): Output[A] = Output.of(x)
3   extension [A](oa: Output[A])
4     def flatMap[B](f: A => Output[B]): Output[B] = oa.apply(f.asJava)
```

The `pure` method is defined with the Java's Pulumi method `of`. It simply boxes the value in an `Output` context.

The `flatMap` function for the `Output[A]` type is implemented using the `.apply` function offered from the Pulumi's Java `Output` class. The `apply` function has a different type from the `applyValue` one. As we have seen above, `applyValue` is matching with the type of `map`, while the `apply` has the following signature: `<U> Output<U> apply(Function<T, Output<U>> func)`. Since the function passed to `apply` takes a type `T` as input and returns a type `Output<U>` as result, and the whole `apply` returns an `Output<U>`, we have a perfect type match with the `flatMap` signature. In fact it will suffice us to use `apply.(f.asJava)` to have the work done.

Thanks to this implementation we are able, as we have seen in the [Subnets creation in Scala](#) paragraph, to use `map` on `availabilityZonesNames()` to apply some logic upon the extracted zone names. In fact, `availabilityZonesNames()` is `Output[GetAvailabilityZonesResult]`, and being now `Output` a monad, the `map` function is supported. We succeeded in getting rid of the `.pulumiall` method.

4.6.6 Automatic code generation for the syntactic sugar

The generation of the syntactic sugar code required the following 2 passages:

- * Analyze the source code of Pulumi's Java APIs for AWS EC2 in order to infer information about the constructors of the various resources and the builders' methods present in the Builder of each **Args** class
- * Use the extracted information to automatically generate the code

The first step has been quite straight forward with the aid of the `JavaParser` library. We'll introduce this Java library in the following section.

The second step has been more problematic. As a first attempt, I tried to use the metaprogramming offered from the new Scala 3 Macros, to create the autogenerated code in the form of an [abstract syntax tree](#) (AST). This solution was potentially promising, since Scala also offers the opportunity to convert these ASTs in code and vice versa at compile time, and so telling us about any error at compile time. The problem encountered is that, once we defined the macros for a new type (like `cidrBlockOwners`), such a type wasn't available at compile time for the other code. In other words, the types generated through the macro, cannot be referenced in the same project since the whole compilation must finish before having the chance to use the brand new types. Another option was represented by `Scalameta`, a library to read, analyze, transform and generate Scala programs, but it isn't compatible with Scala 3, and so had no chance to use it. If the support for Scala 3 will be added to `Scalameta`, it should be considered as a better approach for the generation of the syntactic sugar code in the future.

So, for the second step, a standard *naive* approach has been adopted. The auto-generated code is created with a program that inserts the piece of information extracted from the analysis of the Java APIs with `Javaparser` into a template for the `PulumiBuilderUtilFunctionsForScala` and the `PulumiUtilFunctionsForScala` functions. The generated code can then be exported as a library and included into the dependencies of the building tool used in the Pulumi Scala project.

Pulumi Java APIs libraries inspection with `JavaParser`

Since the code for the inspection is quite verbose (`Javaparser` is a Java library) and not particularly interesting for the aim of the thesis, I won't report any of the code here and I'll limit to describe the steps that are done to infer the required information from the Java libraries for Pulumi.

DirExplorer The first class that I defined is `DirExplorer`. This class has the objective to find all the files in a given directory (and the files in the subdirectories), letting us apply some extra logic during its traverse. We'll be using such a class in a `InferInformation` class to extract all the names of our interest.

InferInformation In the `InferInformation` class we have 3 methods:

listBuilderMethods is the function that opens every `...Args.java` class and, after having parsed an AST of such file, will save all the methods of its builder in a data structure that we'll introduce soon

listConstructorMethods is the function that opens every other file **different** from `...Args.java` and, after having parsed the corresponding AST, checks if a public constructor for the given resource is available. In such a case it will add the name of the class to a List that represents all the constructors that should be generated for our syntactic sugar code.

listFiles is an helper function for the other two methods that just provides the file names of the classes inspected

The data structure used to store the builders' method has the type `Map[String, (String, LinkedList[String])]`. The key of this map is the name of every different builders' method encountered during the parsing of the files. The value of the Map is a list of all the `...Args` classes that contain such a method. In other words, for each method we map all the classes that define such a function. The entry for the `cidrBlock` method on the Map (parsing all the files) looks like this:

```
cidrBlock: [
  DefaultNetworkAclEgressArgs,
  DefaultNetworkAclIngressArgs,
  DefaultRouteTableRouteArgs,
  GetSubnetArgs,
  GetSubnetPlainArgs,
  GetVpcArgs,
  GetVpcPeeringConnectionArgs,
  GetVpcPeeringConnectionPlainArgs,
  GetVpcPlainArgs,
  NetworkAclEgressArgs,
  NetworkAclIngressArgs,
  RouteTableRouteArgs,
  NetworkAclRuleArgs,
  SubnetArgs,
  SubnetCidrReservationArgs,
  VpcArgs,
  VpcIpv4CidrBlockAssociationArgs
]
```

Among all this values, we can find the `VpcArgs`, `RouteTableRouteArgs` and `SubnetArgs` that we used in our implementation, and to which we passed a CIDR block using the `cidrBlock` method.

With the information achieved we are now ready to fill in the templates and generate the syntactic sugar for the Scala APIs of Pulumi.

Raw automatic syntactic sugar code generation

The class that generates all the code is quite simple. A Java `FileWriter` will take care of writing all the strings that represents our code in the `PulumiBuilderUtilFunctionsForScala.scala` and `PulumiUtilFunctionsForScala.scala` files.

We have two functions, named `writeContentForBuilders` and `writeContentForConstructors` that will write all the *various pieces* of generated code into the files using the `FileWriter`. With *various pieces* I refer to the generated types for the methods of the builders, the imports, the conversion functions, ecc.

Finally we have a bunch of functions that fill the various templates to generate the *various pieces* of code. These functions are: `generateTypes`, `generateBuilderMethods`, `generateConstructors`, and `generateImplicitConversionFunctions`. To have an idea of how the filling of a template works let's consider the `generateConstructors` function:

```
def generateConstructors(extractedConstructors: Array[String]) : Array[String] =
  for con <- extractedConstructors
  yield {
    val functionName = con.drop(con.lastIndexOf(ch = '.') + 1)
    val builderCon = con + "Args.Builder";
    "def " + functionName.head.toLowerCase + functionName.tail +
      "(param: String)(init: " +
        builderCon +
        " ?=> Unit, initOpt: (CustomResourceOptions.Builder ?=> Unit) = baseOpts): " +
        con + " =\n" +
        "\tgiven b: " + builderCon + " = " + con + "Args.builder()\n" +
        "\tinit\n" +
        "\tgiven bo: CustomResourceOptions.Builder = CustomResourceOptions.builder()\n" +
        "\tinitOpt\n" +
        "\tnew " + con + "(param, b.build(), bo.build())"
  }
```

Image 4.9: Constructors generation

The template is filled with the variables that represent the name of the constructors, and this is done for each constructor that has been found. Since also the other functions are similar to this one, won't be reported here.

Capitolo 5

Comparison between the languages for Pulumì 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/>.
- Context functions.* URL: <https://docs.scala-lang.org/scala3/reference/contextual/context-functions.html>.
- 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/>.
- Scala trait.* URL: <https://docs.scala-lang.org/tour/traits.html>.
- Union type.* URL: <https://docs.scala-lang.org/scala3/book/types-union.html>.
- Using and Given.* URL: <https://blog.rockthejvm.com/scala-3-given-using/>.