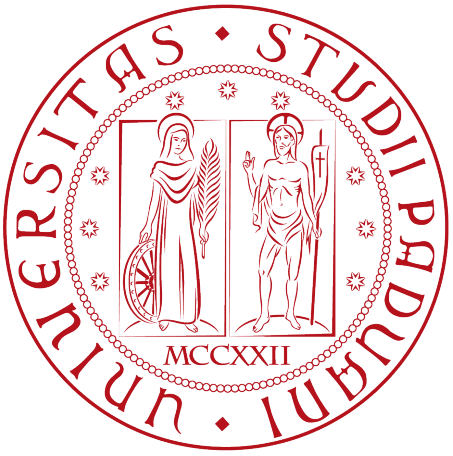
Università degli Studi di Padova

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

Dedicato alla mia famiglia, ai miei amici e a mio nonno che avrebbe voluto vedermi raggiungere questo obiettivo
Dedicated to my family, my friends, and my grandpa who wished he had seen me achieve this

Abstract

Scala as a candidate to address the problem of the always more complex management of the cloud infrastructure

In the last decades the infrastructure used by companies changed deeply. Years ago they were used to have on premise bare-metal servers, running their own applications, while now they rent cloud servers or applications and host their services in a serverless fashion.
Such a modern approach granted enormous benefits to flexibility, but at the same time greatly increased the complexity to manage the configuration of such infrastructures. The current situation requires companies to resort to advanced tools for managing the always more complex structure of the owned cloud resources. The critical aspects of such tools are the range of cloud service providers (AWS, Azure, ecc.) supported and the programming language offered to work with such tools.
Among all the tools that started to appear, Pulumi is the one that results to be the most innovative. With its multi-cloud and multiple general programming languages support, has all the features to address the high requirements imposed by the more and more complex cloud scenario.
This work proves why the addition Scala represents a cool addition to the Pulumi's current pool of supported languages, and how a partial support of the language has been achieved in a very limited time.

Ringraziamenti

Per iniziare, vorrei esprimere la mia gratitudine nei confronti della Prof.ssa Silvia Crafa, relatrice della mia tesi per ben una seconda volta, che ha sempre mostrato disponibilità e cortesia durante quest'ultima parte del mio percorso.

Desidero ringraziare inoltre Andrea Zoleo, che, con il considerevole tempo dedicatomi e i numerosi preziosi consigli fornitimi, mi ha assistito durante lo sviluppo della tesi.

Voglio infine ringraziare la mia famiglia e tutti i miei amici che mi sono stati vicini durante questo mio ultimo percorso accademico, risultando sempre di supporto.

"If you define the problem correctly, you almost have the solution."

— Steve Jobs

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction to the problem

Many years ago, the system management was based on the manual configuration of the company's bare-metal servers. These machines were used to host the various applications and services offered. The configurations for those servers were based on custom shell scripts or even manually achieved through a manual setting based on variables' values read from configuration documents.
With virtualization, the decoupling between software and hardware allowed for scalability of the software on the various bare-metal servers, but added complexity to the configuration due to the virtualization layer that has been added.
Then the trend shifted towards solutions that granted companies the possibility to rent bare-metal servers, from the cloud service providers (CSP), and host virtual machines, internally running the company's applications, on them. It is the birth of the cloud. All this led to the need of having configuration files to ensure repeatability and reproducibility of the operations done on the cloud.
The evolution continues with the advent of the containers, container orchestration, and the virtualization of always smaller applications. This fragmentation of the virtualized services required a further effort in the management of the configuration for the creation and maintenance of always smaller resources. With this the serverless computing begins.
With the serverless model, the need for configuration management tool is made clear. The configuration management tools based on a declarative approach that started to appear are different from each other and can be more flexible on certain aspects, and less in others. Among all the characteristics that we could take in consideration, we shall focus on their possibility to support the management of the resources on multiple cloud providers and if they support markups languages or general purpose languages for the definition of the configuration of the resources. Since the competition of the various cloud providers brought the companies to adopt multi-cloud solutions, picking some services from a certain CSP and others from different CSPs, a tool that allows multi-cloud support is preferred. This would ensure to have a single tool for the management of the entire pool of services hosted on the various cloud providers.
At the same time, the increasing complexity of the cloud and the serverless approach are requiring more and more robust and mature languages for the management of the configuration files for the cloud resources. Amidst the various configuration manage-

ment tools, Pulumi is the only one that provides a multi-cloud enabled solution and at the same time supports many general purpose languages, instead of using the less expressive markup languages.

1.2 The aim of the work

Every programming language has its own features and characteristics, that are making it a better choice for some use cases, and a worse one for some others. Moreover, when choosing a programming language we should also consider all the complementary benefits or shortcomings that it is carrying over. With this I refer to the quality of its building system, the maturity of the available IDEs, the available libraries, the documentation available, and so on.
The aim of the work was to ensure if Scala could be a valuable addition to the current pool of languages supported by Pulumi, that currently are: JavaScript, TypeScript, Python, Go, .NET and (since 2022) Java. Scala has been chosen for two main reasons. First, it is a mature and robust language, with a powerful building system, adopted worldwide, with many libraries, with the interoperability with the JVM and all its others languages, that can be used in excellent IDEs (such as IntelliJ IDEA and Eclipse). Second, but not less important, its functional paradigm and its nature prone to the definition of internal DSLs should allow to create an expressive *syntactic sugar* that enables concise and readable solutions for the configuration files for the management of the cloud resources.
To prove our thesis about the worth of adding a Scala support to Pulumi, a case study based on an AWS EC2 resources generation with Pulumi has been picked. First, a TypeScript solution for the case study has been implemented. Then, the *syntactic sugar* for the Scala support for Pulumi has been coded, to eventually implement the Scala solution of the case study.
The TypeScript solution resulted to be very readable. This was expected since TypeScript has a powerful declarative syntax thanks to its support for the JSON format. Surprisingly from the comparison between the two implementations, despite the more verbose nature of the Scala language, the generation of a concise and readable solution also for Scala has been achieved. Such readability is much appreciated in the declarative approach of the management tools for the cloud resources. These configuration files must be kept clear and must be easy to read to ensure to ease any future management of the infrastructure.
Scala, differently from TypeScript, has a powerful build system, allows for an excellent management of the code thanks to the packages, and the development is supported by powerful IDEs. So it is representing a cool addition that, with respect to TypeScript, can better exploit the refactoring, reuse, and the logical organization in a maintainable structure of the code for our configuration. And this is perfectly fitting the high requirements coming from the always more complex serverless based cloud scenario.
As extra work, the automatic generation of the *syntactic sugar* code for a partial Scala support for Pulumi has been achieved.

Chapter 2

Introduction to Infrastructure as Code

An introduction to the Infrastructure as Code (IaC). The birth of the IaC is firstly presented, then the possible approaches, the advantages of using IaC, its challenges, and the evolution of the IaC with the second generation of tools. Finally the related works are presented

2.1 How we arrived to the IaC

2.1.1 Bare-metal servers and manual configurations

Originally companies owned bare-metal servers on which hosted own services. Each server was capable to host a single service. The configuration of such servers was manually made using paper sheet with information about the configuration parameters (such as IPs, environment variables, etc.), or with shell scripts that had to be manually maintained as the infrastructure changed or had to be reconfigured. Some templates have been created to slightly automate the process of reconfiguration of the servers.

2.1.2 The advent of the virtualization

The virtualization era began as the virtual machine concept was born. Here companies still had to own bare-metal servers, but the virtualization granted the possibility to scale software services upon one or more hardware servers. This was possible to the decoupling between the hardware and the software. However, to achieve this goal a further amount of configuration was required to be given and maintained. The virtualization layer is having its own configuration, on top of the one of the bare-metal server.

2.1.3 The cloud

For a company, buying its own hardware servers have an extremely high cost at the beginning. With the virtualization, the possibility to borrow hardware servers was

made possible. The virtual machines of the company are run on the rent servers thanks to the virtualization. These operations were made possible by the management console offered by the providers of the servers, that allowed to instantiate the application as a virtual machine on their servers.
With this cloud service providers were born, offering the bare-metal server rental service with a pay as you go pricing. The more resources you use (or rent), the more you pay. This mechanism led to the birth of the cloud.
Such an approach zeroes the initial costs to both buy, setup, and maintain a personal infrastructure. Obviously this led to the need of having configuration files to ensure repeatability and reproducibility of the operations done on the cloud. The engine that interprets such files is host on the cloud itself.
At the same time, the APIs used from the management consoles have been exposed as official automation APIs. We'll see soon how such APIs have a critical role in the IaC scenario.

2.1.4 Towards the serverless

A further step in the virtualization scenario has been achieved when the concept of container was born. Before containers, whole virtual machines were being run on the virtual servers. In most of the cases, the virtualization of the whole operative system (OS) of the virtual machine was an extra useless overhead. The containers permitted to instantiate a virtual instance of service or application without the extra overhead of the OS.
Along with the concept of container also the container orchestration was born. Such services allow containers to inter-operate. In fact, the fact that containers contain single applications or services, they will likely need to communicate with each other. Kubernetes for example, a container orchestration tool, is allowing to configure the interfaces to let containers communicate to each other.

2.1.5 Serverless trend

The most common applications started to be directly provided by the cloud service providers. At this point, the user didn't have any more to take care of virtualizing a virtual machine or a container on a specific server. He just had to request for the desired resource to the cloud provider and it would have taken care to virtualize it on an available machine (that could have been different any time the service was to be made available).
With this trend, also smaller custom logic started to be executed on virtual servers, such as custom lambdas, queues of messages, cache memory services, etc.
All this led to the serverless computing, which has put even more pressure on the configuration required for the infrastructure.

2.1.6 The advent of the Infrastructure as Code

We notice from the previous paragraphs that, over time, the amount of configuration required has kept increasing over time, reaching its peak with the current serverless cloud scenario. The necessity of configuration management tools was becoming real to keep up with the increasing complexity of the cloud structure and functioning.
With the advent of such tools, the Infrastructure as Code was born.

2.2 Infrastructure as Code

2.2.1 What is IaC

The evolution we have seen before has brought to an engineering of the system administration operations. The management of configuration files for the infrastructure as code is at the base of the Infrastructure as Code (and here is why it is called so). In other words, we could call Infrastructure as Code the process of managing and provisioning IaaS through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.
The IT infrastructure managed by this process comprises, as we have seen in the previous section, both physical equipment, such as bare-metal servers, and also virtual machines, virtualized applications and services, and the 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.
In the How we arrived to the IaC section we have seen how first the APIs of the cloud service providers have been exposed, and only after the configuration management tools were born. The exposed APIs are by nature imperative. In fact their usage is cumbersome and the user must be aware of what he is actually doing, otherwise configuration errors might happen.
The configuration management tools that were born after the advent of the serverless scenario are instead using a declarative approach.

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.

2.2.2 Infrastructure as Code tools' main parts and their functioning

From now on, for the aim of this thesis, with IaC tools (or configuration management tools) we'll be referring to set of the following technologies: GCP, Azure, CloudFormation, Terraform, CDK, and Pulumi.
The various existing IaC tools have a different architectures and functioning, but at their base they share the same abstract structure:

declarative APIs for the user these APIs let the user declare the resources that he would like to instantiate on a certain cloud service provider

backend engine this engine will look at the resources declared by the user through the user APIs and will convert such declarations to REST API calls to the cloud service provider to instantiate them

Obviously each tool has its own much more deep functioning. We'll see how Pulumi works in the Pulumi functioning section.

2.2.3 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. Being code, infrastructure prototypes can be programmed and shared across the various teams of the company, boosting speed and reducing costs even more. This feature makes your code base less verbose and more readable while at the same time team members are encouraged to apply best practices.
Moreover, 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.
Related to this, if we create all the resources of the infrastructure using a program, we can always know what are the resources that we have on the cloud. On the other hand, before IaC, once the administrators created the desired resources on the service providers, there was no way to know which resources were actually present. Tools to inspect the state of the cloud providers were created, but their reliability is not bulletproof. The modern IaC approach, and especially with advanced tools as Pulumi, addressed this problem masterfully since the entire declaration of the infrastructure is coded in single and a well-structured project.
Finally, when automation functionalities are provided with the considered IaC tool, we are given the possibility to define various branches (develop, stage, production, etc.). This was not possible without IaC, the infrastructure was one and only one. The chance to have different branches let us test and experiment the changes of our

infrastructure before actually bringing them on stage and eventually on production.

2.2.4 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.

2.2.5 Evolution of IaC tools

Tools designed for Serverless Applications - the first wave

The foundation of IaC in the public clouds is these three cloud vendor-specific IaC tools: 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 enable building and testing your serverless applications locally on your machine.
Some of these tools made a further step by supporting multiple cloud service providers in a single solutions, like Terraform.

Serverless Infrastructure as programming language code - the second wave

Declarative languages have 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.

2.3 Related works

Pulumi is quite new as IaC platform, how did it manage to emerge in a scenario where many other IaC technologies were already present? Let's first introduce the scenario present when Pulumi showed up.

2.3.1 Pulumi and the other solutions

The two main aspects of IaC tools: supported languages and cloud providers

A IaC tool can have its own scripting language to define the resources, or, it can let the user choose from many general purpose programming languages.
The scripting languages, as we already mentioned, are a distinctive feature of all the first generation IaC tools. These tools comprehend for example Terraform and CloudFormation.
New tools then showed up, allowing the user to pick from a pool of general purpose programming languages when interacting with such a technology. For these case we'll mention AWS CDK and Pulumi.

Another distinctive trait among the various technologies is the range of supported cloud providers.
The cloud providers developed their own IaC tool to manage the cloud resources they were offering.
Then, some innovative IaC tools have been able to manage resources on different cloud providers. This is clearly a convenient functionality since it lets the user have a single tool to manage all the cloud resources coming from various different providers. Both Pulumi and CDK let the user choose from many commonly used programming languages.

For how we have seen in the How we arrived to the IaC section, the increasing complexity of the infrastructure with the advent of the cloud and the serverless requires more and more robust tools and programming languages to exploit at the best the refactoring, reuse, and the logical organization of the code in well defined structures or packets.
Moreover, the competition that is involving all the various providers is bringing the companies to choose multi-cloud solutions. Because of this, IaC tools that support multiple clouds are preferable to the ones supporting a single cloud service provider.
Let's now see how Pulumi and the other various available tools are satisfying these two main aspects.

CloudFormation: scripting languages and a single cloud provider supported

As a first wave IaC tool we have CloudFormation. It's one of the founders of IaC but being based on JSON/YAML files for defining resources and working only with AWS has a limited flexibility.

Terraform: scripting languages and multiple cloud providers supported

Terraform proposed as a solution to manage multiple resources coming from different cloud providers with just one tool. The struggle for companies to adopt many different IaC tools, one for each cloud provider, was not negligible. Such a solution allows to cut costs and efforts to manage the cloud resources of the various infrastructures.
Anyway Terraform is based on the HTC scripting language, limiting its expressiveness in the possible IaC solutions.

CDK: general purpose programming languages support for AWS

Orthogonal to Terraform we have CDK. This technology, with respect to TerraForm, traded the multi-cloud support for a multi-language support. The rich pool of supported languages (TypeScript, JavaScript, Python, Java, C#/.Net, and Go) allows the user to choose the favorite programming language. All the features of the selected programming language can boost the offered solutions for the infrastructures, but at the price of sticking to the AWS cloud provider.

Pulumi: general purpose programming languages and multiple cloud providers supported

Pulumi doesn't want to give up on anything. It succeeded in offering a solution that is flexible under both of the two aspects. Pulumi managed to achieve what the other IaC tools and platforms did not, and because of this is bringing on the table some innovation and added value with respect to the competitors.
Pulumi successfully abstracted from the problem in order to find a smart solution to achieve both the multi programming language support and the multi-cloud service provider support. It decoupled the REST APIs offered from the various cloud service providers (to create the resources) from the user APIs of Pulumi (that are used from the user to inform Pulumi about what resources should be created on the various CSPs). Pulumi's backend engine converts the code written from the user to effective REST API calls to the various cloud service providers, and eventually create the resources.
This workflow is the major innovation brought by Pulumi, and for sure the key that allowed it to don't give up neither on the flexibility of the supported programming languages and nor on the supported cloud providers.

2.3.2 Java addition to Pulumi's pool of languages

All the hidden advantages of the Java support for Pulumi

Java support has been added to Pulumi during the 2022. The large amount of work done by the Pulumi's team has brought advantages from various points of view.
When we choose what language to use from the various opportunities offered by Pulumi,

we cannot limit our choice to the features offered by the language and our tastes. We shall also consider the robustness of the building system, the available libraries, the code management, the documentation, the user base, and the available IDEs for that language.
For example, nodejs (TypeScript's building system) is less mature and robust with respect to the Java's one.
Moreover, the Java development features a good amount of mature and complete IDEs. IntelliJ IDEA and Eclipse, to mention the most famous ones, are powerful IDEs that allow the user to do effective refactoring, inspection, reverts, and more on the code.
The features of the language sometimes are also affecting what characteristics an IDE can have. For example, TypeScript's duck typing is not of help to the IDEs when it comes to code inspection and refactoring. On the other hand, the much more robust and well defined type system in Java is granting IntelliJ and Eclipse amazing tools to manage the code at our please.
We could mention also how Python and Java have many libraries to choose from to enrich our solution. TypeScript instead is having much less.
Furthermore, Java has a much better management of the code with respect to TypeScript. The Java packages are useful when the solution grows in size, while TypeScript's one will eventually start to "creak" due to its poor code management features. An IaC architecture will keep growing in size as time passes, so the possibility to use a programming language that offers good tools to manage the codebase, such as Java, is a valuable feature to keep in mind when choosing the programming language.
Last but not least, with Java addition to Pulumi, all the languages based on the JVM can actually work with Pulumi. In fact the great effort made from Pulumi is actually opening the doors to many more languages such as: Scala, Kotlin, Groovy, Clojure, etc. In fact, my Scala solution has been made possible exactly because of this fact, since I defined my Scala APIs for Pulumi on top of the Java APIs.

The onerous work to officially support a new language in Pulumi

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. The full procedure can be found on New Language Bring Up.

Chapter 3

Pulumi, an IaC platform

An introduction to Pulumi, to its advantages and its core functioning

3.1 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 is innovative because, differently from most of all the other second generation IaC tools, has been able to abstract the problem of having many different cloud providers, and also of really different natures (such as Kubernetes vs AWS). Its solution is letting the user rely on a single IaC platform to manage any kind of resource on any cloud provider. Potentially, Pulumi could support any technolgy that is showing REST APIs. This feature is making Pulumi an innovative IaC platform that rise IaC on a new level.
Pulumi offers a range of tools and features to simplify the development and management of cloud infrastructure, including version control, testing, 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.

3.1.1 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).
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.

3.2 Pulumi functioning

3.2.1 Overview of the functioning

Pulumi functioning is based on the interoperation of three main parts:

Source code is used to declare the resources to be created on the respective cloud service provider

Backend engine such engine could run on the Pulumi server or locally on the machine. When the `pulumi up` command is executed, a state file is created and stored in the backend engine. Such a file is representing a "screenshot" of the resources declared in the user's project. The engine then uses such a file to create the descripted resources on the cloud providers, thanks to the REST APIs that they are exposing

Resources of the providers are the actual resources on the cloud providers

This image, taken from the official site of Pulumi, is representing the aforementioned architecture.

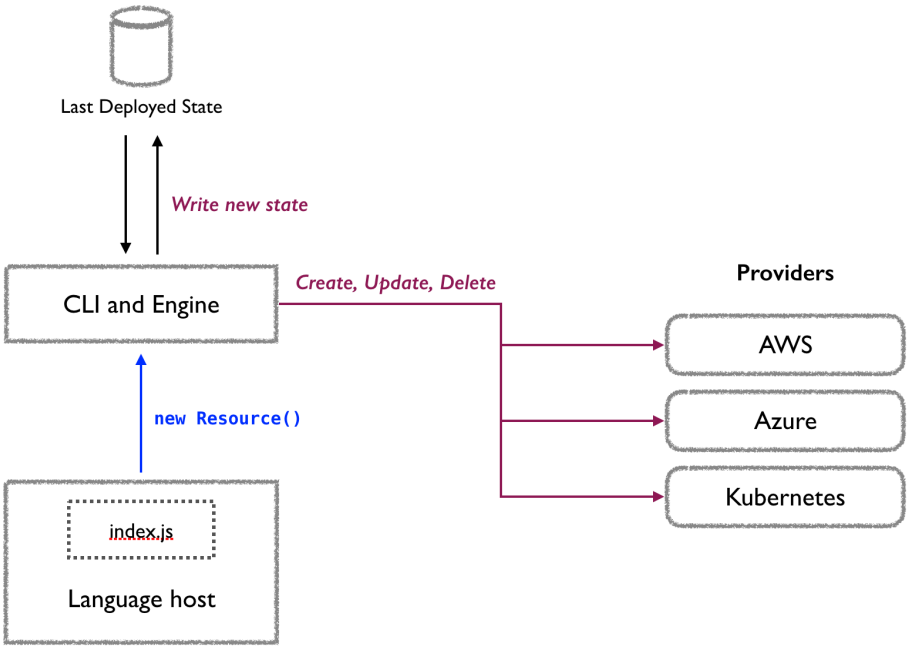


Image 3.1: Pulumi architecture

The resources on the cloud providers could drift and their state might get different from the logical description in the state file of the backend engine. This could happen if an administrator manually modifies the resources without passing through the Pulumi project, or also if the cloud provider changes the APIs on an update. The `pulumi refresh` command is telling the engine to check if the states on the cloud providers are corresponding to the logical representation of the state file. If it is not the case, the resources are modified in order to be aligned with the state file.

3.2.2 Pulumi project

The interesting part for the user is the Pulumi project. Such a project, that can be created with the `pulumi new` command, can be any folder that contains a Pulumi.yaml file. This file specifies the runtime to be used (nodejs, pyhton, Java, etc.) and where to look for the program that should be executed during deployments.
The execution of the program will, through the Pulumi APIs, call the Pulumi engine to create the desired resources on the respective cloud providers.

3.2.3 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 an environment. Tipically, we have a stack for the develop, one for the stage and one for the production. This is granting us the chance test safely our infrastructure before pushing the changes on production.

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*.

3.2.4 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 the respective resource constructor functions. This function takes in the configuration for the resource as input, and returns a reference to the resource to be created.
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.

3.2.5 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.
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.

3.2.6 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.

3.2.7 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.

3.3 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 in Java, or a Promise in TypeScript, 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.

Chapter 4

Scala: modern, functional and object-oriented

Scala brief overview. First the language paradigms are introduced, and then the functionalities used for the thesis are introduced and explained

4.1 Introduction to Scala

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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 base of the functional programming paradigm since it lets concatenate functions and create more complex and more powerful abstract constructs, such as functors, applicatives, and monads, to eventually work on data in an immutable and 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 chance of taking functions as parameters, will be a key ingredient for the highly expressive and readable solution achieved and proposed in the AWS EC2 resources 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 AWS EC2 resources 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 type matching to the implicit parameter's one, if present in the scope, will be automatically injected in **at compile time**.
Lets consider an example taken from Given and Using Clauses in Scala 3 - Rock the JVM to better understand such concepts:

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

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

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

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

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

This is a key functionality that, among with other Scala features, will let us achieve the goal of the thesis.

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. 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 lets consider a brief example:

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

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

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

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).

Chapter 5

AWS EC2 resources 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

5.1 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.

5.1.1 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.

5.2 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

5.2.1 Components of the infrastructure

The infrastructure I 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:

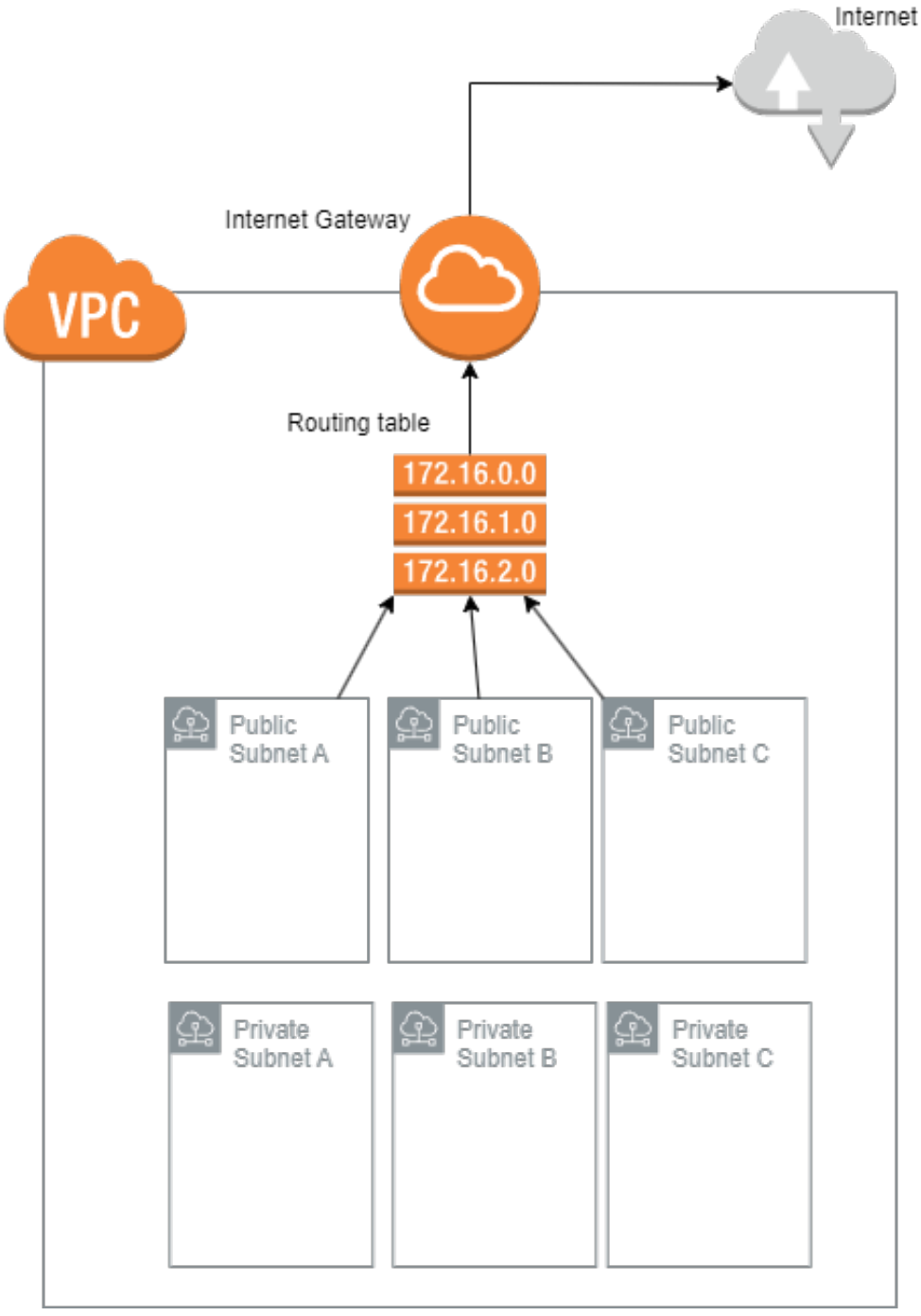


Image 5.1: Infrastrutcture Architecture

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 I 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.

5.3 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.

5.3.1 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.

5.3.2 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.

5.3.3 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:

```
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:

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

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:

```
protected createAZsSubnets(isPvt: Boolean) : Output<Subnet[]>{
  this.availableZones = aws.getAvailabilityZonesOutput()
  return pulumi.all([this.availableZones.names, this.vpc!.id]).
      apply(([azNames, vpcId]) => {
    let i = 0
    let listToPushInto: Subnet[] = Array<aws.ec2.Subnet>()
    azNames.forEach(azName => {
      let fullName = azName + (isPvt ? "-pvt" : "-pub") + "-
          subnet-typescript"
      listToPushInto.push(new Subnet(fullName, {
        vpcId: vpcId,
        availabilityZone: azName,
        cidrBlock: isPvt ? this.pvtSubnetsCidrs[i] : this.
            pubSubnetsCidrs[i],
        tags: {
          Name: fullName,
        },
      },{
        parent: this.vpc
      }));
      i++;
    });
    return listToPushInto
  });
}
```

In the code we instantiate private or public 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 creation of a subnet requires us to provide both the availability zone name and the ID of the VPC. Since these two pieces of information are contained in two separate `Outputs` (`Output<String[]>` and `Output<String>` respectively) we must first use the `pulumi.all` to wrap them in a single `Output<String[], String>`. This due to the fact that `.apply` can accept a single `Output<A>`, for any `A` type, value as input and not a pair. Then, we can use the `.apply` to unbox from the `Output<String[], String>` its inner value. Now that we have both the pieces of information out of the *context* value, we can use them to create the subnets.
We can notice that the `.apply` function is, at the end of its lambda function, returning a list of the created subnets. Such a return is also the return of the function and is indeed of the `Output<Subnet[]>` type.

5.3.4 Routing table creation

This is the code to create the routing table resource:

```
protected createRouteTable() {
  this.routeTable = new RouteTable("example", {
    vpcId: this.vpc!.id,
    routes: [
        {
            cidrBlock: "0.0.0.0/0",
            gatewayId: this.gw!.id,
        },
    ],
    tags: {
        Name: "myRouteTable-typescript",
    },
  },
  {
    parent: this.vpc,
  });
}
```

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.

5.3.5 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:

```
protected attachRouteTableToPubSubnets(){
  let i = 0
  this.pubSubNets.apply(subNets => {
    subNets.forEach(sn => {
      new aws.ec2.RouteTableAssociation(`${i}-
          routeTableAssociation-typescript`, {
        subnetId: sn.id,
        routeTableId: this.routeTable!.id,
      },
      {
        parent: this.vpc
      });
      i++
    })
  });
}
```

Here we used once more, as for the subnets creation, a combination of `.apply` and a foreach to iterate over all the extracted subnets (`.foreach`) from the `Output` *context* (`.apply`).
Inside the `foreach`'s lambda 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.

5.4 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:

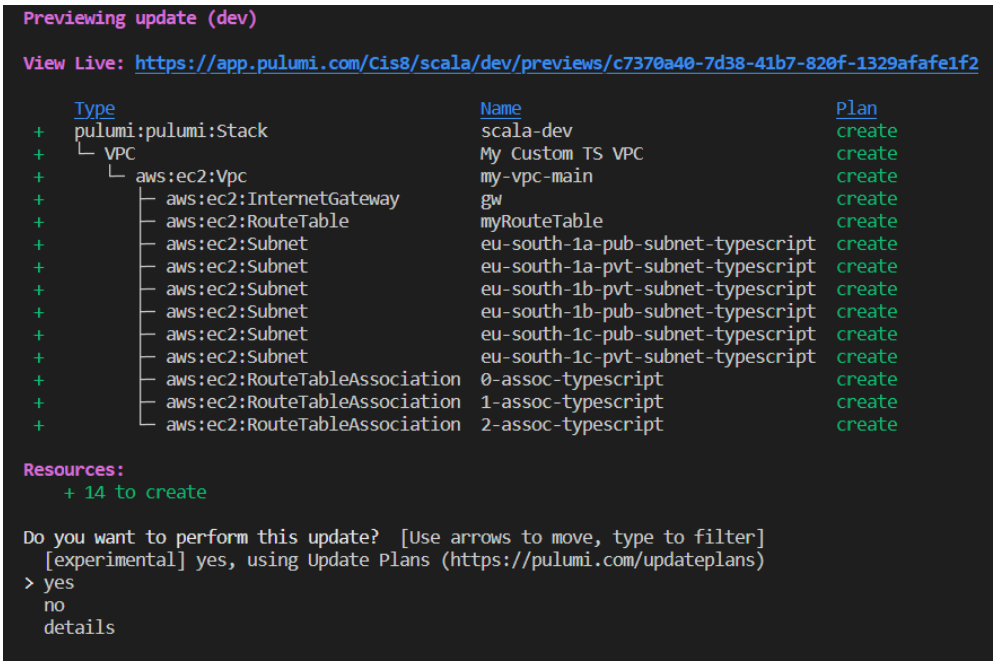


Image 5.2: pulumi up preview

If we press yes this is the output:

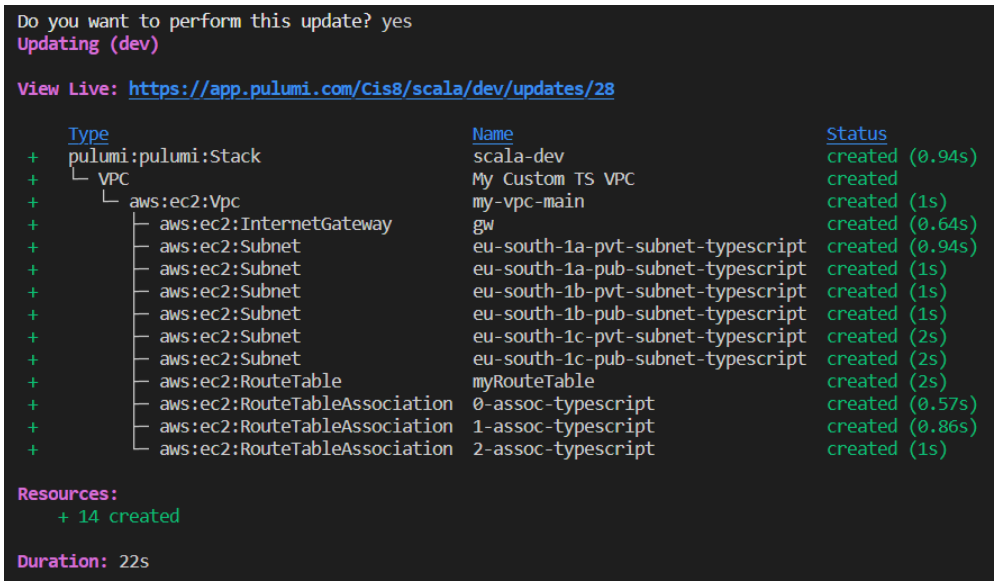


Image 5.3: pulumi up confirmed

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 stuck nicely ordered and tied up.

5.5 Destroying the resources with Pulumi

Now lets 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 5.4: 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 5.5: pulumi destroy confirmed

5.6 My Scala implementation of the case study

Since Scala is not supported by Pulumi, I had to implement it on my own. As wI mentioned in the The onerous work to officially support a new language in Pulumi section, an official support of Scala for Pulumi was unfeasible, hence a custom and partial support of Scala has been achieved. 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 *sugared* 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 if 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 presented.

5.6.1 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. Lets consider for example the Vpc resource. In Java, to instantiate such a resource we'd call:

```
protected Vpc vpc = new Vpc("my-vpc-java", VpcArgs.builder()
  .cidrBlock("10.136.0.0/24")
  .tags(Map.of("Name", "main"))
  .build(),
        CustomResourceOptions.builder()
                .parent(this)
                .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.

5.6.2 *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 *sugared* function that represent the constructor of that resource, and 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("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 *sugared* 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 `VcpArgs` 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("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     for
3       azRes <- availabilityZonesNames()
4       myVpcId <- myVpc.id()
5       tuples = azRes.names().zip(if isPvt then pvtSubnetsCidrs else
            pubSubnetsCidrs)
6     yield
7       tuples.map((name, cidr) => {
8         val fullName = name + "-" + (if isPvt then "pvt" else "pub") +
              "-subnet-scala"
9         subnet(fullName) ({
10          vpcId(myVpcId)
11          availabilityZone(name)
12          cidrBlock(cidr)
13          tags("Name" -> fullName)
14        },{
15          parent(myVpc)
16        })
17    })
```

We can notice how we achieved to get a solution that is relying only on the for yield construct thanks to the monad for the `Output` type, that we will soon see how it is actually implemented.
The `azRes` enumerator is extracting a `GetAvailabilityZonesResult` object out from the `Output[GetAvailabilityZonesResult]` object returned by `availabilityZonesNames()`, and the `myVpcId` is instead extracting the id of the Vpc from the `Output[String]` type coming from `myVpc.id()`. **These two enumerators are the replacement of the `pulumi.all` function in the TypeScript solution.** These extractions are possible only thanks to the mondadic implementation of the `Otput` type. In fact this

syntactic sugar of the for yield is, behind the scenes, implemented as a concatenation of map and flatMap functions, that are exposed by the `Monad[Output]` type.
At line 4 we have the definition of the `tuple` value as a zipping of the availability zone names (extracted with the `.names()` function) and the respective CIDR blocks. The tuples are mapped with a lambda that declares the various subnets to create. Now, the yield is concatenating the various declared subnets in a single `Output[Iterable[Subnet]]`, that is the return type of the for yield and so also of the function.
We should in fact observe the fact that the for yield takes an `Output[A]` and returns an `Output[B]`, where `A = GetAvailabilityZonesResult` and `B = Output[Iterable[Subnet]]`, that is resembling to the map input/output types.

Attaching the subnets to the routing table

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

Similarly to the subnet creation function, also here we use the for enumerators to "unbox" an `Output[Iterable[Subnet]]` value. After having zipped the subnets to the indexes, whose only purpose is to give a custom name to the created associations, we iterate over the tuples to declare the associations with the aid of a map function.
Since the logic is analogous to the `createAzSubnets` function 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.

Resources creation with pulumi up

Now lets make sure that the *stack* created with `pulumi up` is the same of the one created with the TypeScript implementation. From this image we can see that they are equivalent to the ones shown in Resources creation with pulumi up in TypeScript:

```
Please choose a stack, or create a new one: dev
Previewing update (dev)

View Live: https://app.pulumi.com/Cis8/scala/dev/previews/d2916f08-b77f-4161-83d4-fc2f9323e714

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

Resources:
    + 14 to create

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

View Live: https://app.pulumi.com/Cis8/scala/dev/updates/32

     Type                                 Name                          Status
 +   pulumi:pulumi:Stack                  scala-dev                     created (1s)
 +   └─ VPC                               My Custom scala VPC           created
 +      └─ aws:ec2:Vpc                    scala-main                    created (1s)
 +         ├─ aws:ec2:InternetGateway     gw                            created (0.62s)
 +         ├─ aws:ec2:Subnet              eu-south-1a-pvt-subnet-scala  created (1s)
 +         ├─ aws:ec2:Subnet              eu-south-1a-pub-subnet-scala  created (1s)
 +         ├─ aws:ec2:Subnet              eu-south-1b-pvt-subnet-scala  created (1s)
 +         ├─ aws:ec2:Subnet              eu-south-1b-pub-subnet-scala  created (1s)
 +         ├─ aws:ec2:Subnet              eu-south-1c-pub-subnet-scala  created (2s)
 +         ├─ aws:ec2:Subnet              eu-south-1c-pvt-subnet-scala  created (2s)
 +         ├─ aws:ec2:RouteTable          myRouteTable                  created (2s)
 +         ├─ aws:ec2:RouteTableAssociation 0-assoc-scala               created (0.47s)
 +         ├─ aws:ec2:RouteTableAssociation 1-assoc-scala               created (0.82s)
 +         └─ aws:ec2:RouteTableAssociation 2-assoc-scala               created (1s)

Resources:
    + 14 created

Duration: 22s
```

Image 5.6: pulumi up result with the scala implementation

5.6.3 *syntactic sugar* for the constructors of the resources

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

```
def vpc(param: String)
        (init: VpcArgs.Builder ?=> Unit,
         initOpt: (CustomResourceOptions.Builder ?=> Unit) =
             baseOpts): Vpc =
         given b: VpcArgs.Builder= VpcArgs.builder()
         init
         given bo: CustomResourceOptions.Builder =
             CustomResourceOptions.builder()
         initOpt
         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 `VcpArgs.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:

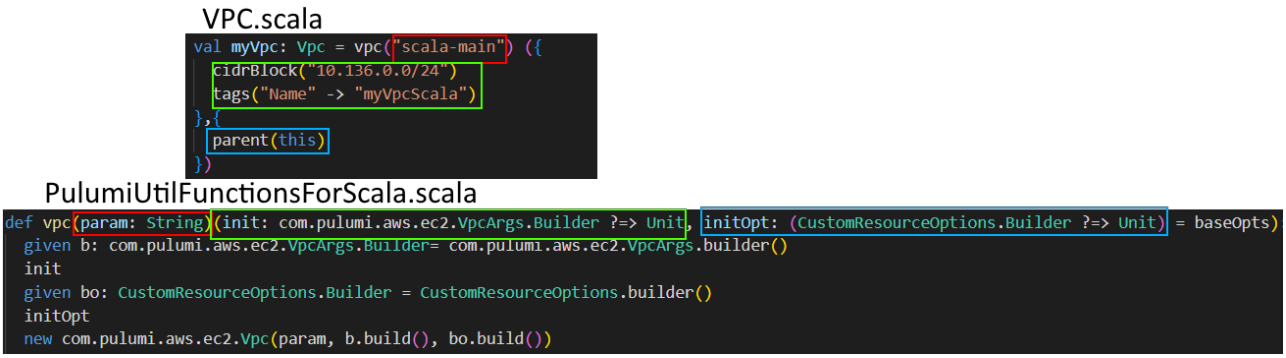


Image 5.7: Parameters correspondence in the VPC declaration in Scala

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:

```
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*:

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

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:

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

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) : etc.
```

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*.

5.6.4 *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 the methods at line 2 and 3 of this code:

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

Once the enclosing 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 bot 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:

```
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. Lets 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:

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

And now lets define another function that is working for the `RouteTableRouteArgs.Builder`:

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

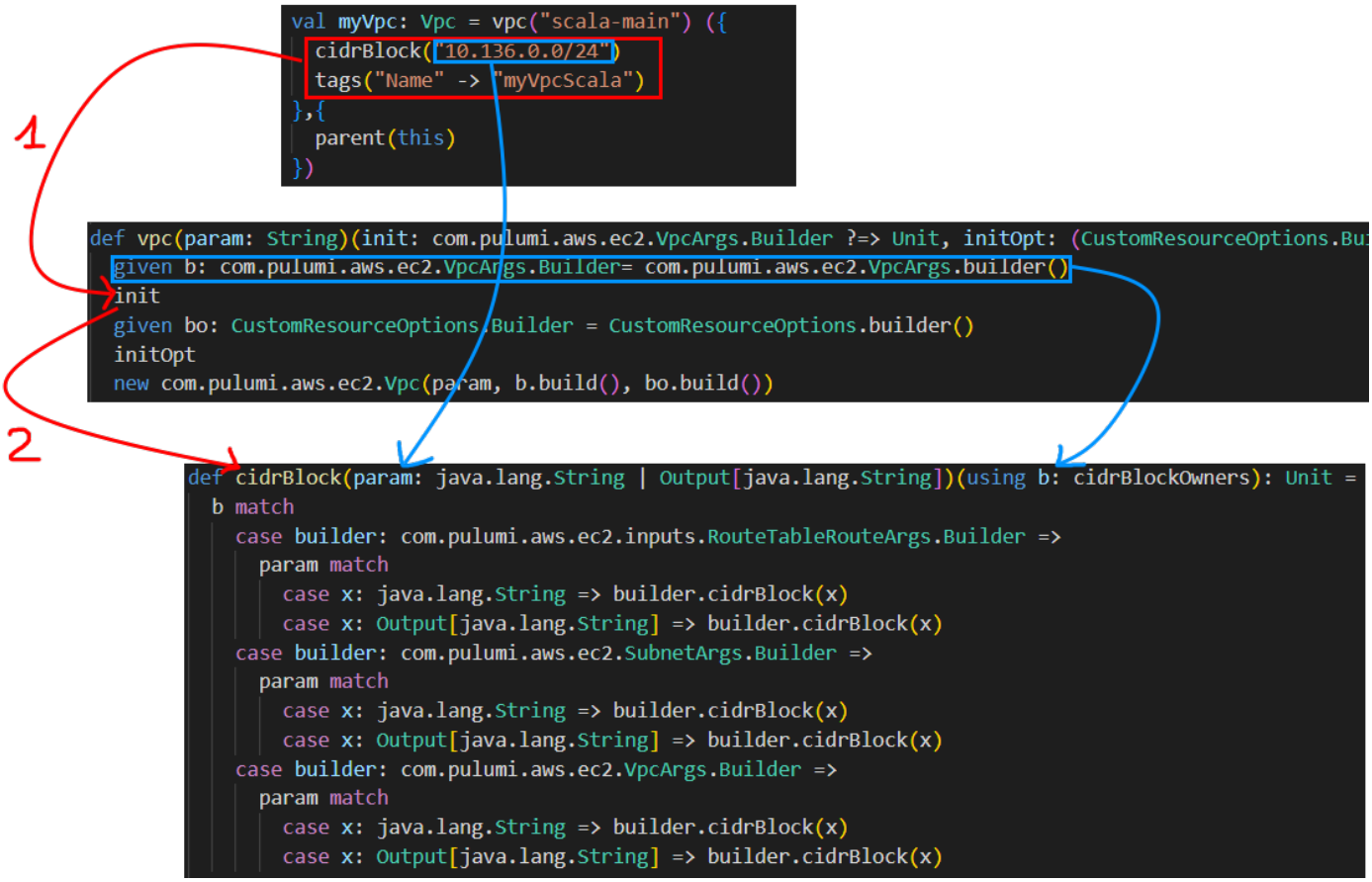


Image 5.8: Function calls flow and parameters passing for the VPC declaration

We can see how the lambda with the calls to our defined methods `cidrBlock` and `tags` is passed as the init parameter of the *sugared* `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 this functions is to achieve the tricks that I mentioned previously (in the Vpc creation in Scala and in the 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:

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

We can notice that the function that converts our tuple into a singleton Map is based on the Converion 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:

```
given elemToList[A <: ResourceArgs]: Conversion[A, List[A]] =
        (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.

5.6.5 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 `Ouput` type has been achieved.
Since a monad is also a functor, lets see how `Functor[Output]` has been implemented.

Functor implementation

First a `Functor` trait has been defined:

```
trait Functor[F[_]]:
  extension [A](x: F[A])
    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:

```
given Functor[Output] with
  extension [A](oa: Output[A])
    def map[B](f: A => B): Output[B] =
      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:

```
trait Monad[F[_]] extends Functor[F]:
  // The unit value for a monad
  def pure[A](x: A): F[A]

  extension [A](x: F[A])
    // The fundamental composition operation
    def flatMap[B](f: A => F[B]): F[B]
    // The ‘map‘ operation can now be defined in terms of ‘flatMap‘
    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 `faltMap`, 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:

```
given Monad[Output] with
  def pure[A](x: A): Output[A] = Output.of(x)
  extension [A](oa: Output[A])
    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 suffices us to use `apply.(f.asSJava)` 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.

5.6.6 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 *sugared* code

The first step has been quite straight forward with the aid of the JavaParser library. The second step has been more problematic.

Scala 3 *macro*s

As a first attempt, I tried to use the metaprogramming features offered from the new Scala 3 *macro*s 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 was that once we defined the *macro* 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*, can't be referenced in the same project since the whole compilation must be finished before having the chance to use the brand new types.
I'll report here the work done that is the attempt to define a custom type to be later used in the same project. In particular, the type definition was trying to address the generation of the custom types, like `cidrBlockOwners`. These types were to be used later, in the same project, with other *macro*s that would have represented the builders' methods and the constructors for the resources.

What is a *macro* Thanks to the useful introduction provided by Pawel L. on Medium, I'll report here the key concepts of the Scala 3 *macro*s. A Scala 3 *macro* is a piece of code executed **by the compiler**. With the *macro*s, it is possible to analyze and generate code. If an expression of type `T` should be used by the *macro*, it needs to be converted to `Expr[T]` — the representation suitable for the *macro*. The process is called *quotation*. The output code representation created in the *macro* is converted back and embedded in the program in the process of *splicing*.
Scala 3 *macro*s manage the code as an AST. To be precise the Scala 3 implementation of the AST is used, that is called Typed Abstract Syntax Tree (TASTy).
A *macro* in general is defined by two functions. One is used to call the *macro* in the code and the other is used define its implementation.
Now lets consider a *macro* to define a new type with a custom name and that represents the `String | Boolean` union type. The function that implements the *macro* is the following one:

```
1   def defineNewTypeImpl(name: Expr[String], types:
2         Expr[Seq[String]])(using ctx: Quotes) : Expr[Any] =
3     '{ type $name = String | Boolean }
```

The *quoting*, identified by the `'{...}` syntax, is encapsulating the code contained within the curly brackets in a TASTy representation. We can also note the `$name` syntax within the curly brackets. In fact, since the `name` parameter is of type `Expr`, and hence is represented as a TASTy, we have to convert it to code in order to use it inside the curly brackets along with the rest of the code. To achieve this we used the splicing: `$`.
Now lets consider instead the function that we'll use to call the *macro*:

```
1   inline def defineNewType(inline name: String, inline types:
       String*): Any =
2     ${ TypeGenMacroImpl.defineNewTypeImpl('name, 'types) }
```

The first thing we can notice from such a function is the presence of the `inline`

keyword. The *inlining*, in Scala, is the replacement of the code in the place of the usage instead of its reference.
So when we'll call this function, its body will be replaced by the compiler at the site of the function call. Since the body of this function is the `splicing` of the call at the *macro* implementation, the code that will be replaced will be in truth the one returned by the definition of the *macro*.
We evince that the key to work with *macro*s is the combo between the *inlining* and the *quoting/splicing*. The final result will be the compilation of a code that has the body of the definition of the *macro* in place of the function call.

What I wanted to achieve with all this was the possibility to define a new type through a *macro* and reference that type somewhere else in the project. So I created this object for the definition of a new type:

```
1  object NewTypeDefinition {
2    defineNewType("MyType", "")
3  }
```

With this code we are actually asking the compiler to replace the code of the `defineNewTypeImpl` function, with `"MyType"` as input parameter, in the place of call of the function at line 2 of the block of code just presented.
The next step was to reference the newly created type somewhere else in the code, in this way:

```
1  @main def hello: Unit =
2    val x: MyType = "MyString value" \\ comp. error
```

Sadly this is not possible because of the problem that has been explained at the beginning of this section. The `MyType` type generated through the *macro* is not available for us until the compilation of the whole project is complete. Hence, where we are trying to define the value `x` of type `MyType`, the compiler complains with the error "`Not found: type MyType`".
This problem caused the whole *macro* approach to fail.

Scalameta

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 I 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.

A naive approach as solution

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

Non Args and Args classes In the Java APIs of Pulumi we have two kind of files (and so classes). The firsts are the **non** ...`Args` files, that are those files that contain the API for the constructor of a given resource. The second kind of files are represented by the `Args` files. Thes classes contains the Builder definition of the respective class and the APIs of the builders' methods used to pass the arguments of the resource while building it with an instance of a builder For example, we have the Vpc.java file and the VpcArgs.java files.

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 the files of our interest.

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

listBuilderMethods is the function that opens every ...`Args.java` file 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 **non** ...`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 worlds, 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, etc.
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 lets 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.toLower + 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 5.9: Code for automatic generation of the resources constructors

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.

The automatically generated *syntactic sugar* code

Since the generated files contains many hundreds of lines of code, I'll report here only some samples of each file.

PulumiBuilderUtilFunctionsForScala First we have some default imports:

```
1  package com.cisotto.myvpc.builder
2
3  import com.pulumi.Context
4  import com.pulumi.Pulumi
5  import com.pulumi.core.Output
6  import com.pulumi.resources.{CustomResourceOptions, Resource}
7  import scala.collection.JavaConverters._
8  import collection.convert.ImplicitConversionsToScala.`
       collection AsScalaIterable`
9  import scala.compiletime.ops.boolean
10 import scala.compiletime.ops.string
11 import scala.language.implicitConversions
12 import com.pulumi.resources.CustomResourceOptions
13 import com.pulumi.resources.ResourceArgs
```

Then we have some all the generated union types, that look like:

```
1  ...
2  type vpcIdOwners = InternetGatewayArgs.Builder |
       RouteTableArgs.Builder | SubnetArgs.Builder
3  type gatewayIdOwners = RouteTableRouteArgs.Builder |
       RouteTableAssociationArgs.Builder
4  type localGatewayIdOwners = RouteTableRouteArgs.Builder
5  type tagsOwners = InternetGatewayArgs.Builder |
       RouteTableArgs.Builder | ubnetArgs.Builder | VpcArgs.
       Builder
6  ...
```

The implicit conversion functions are then printed:

```
1  given tupleToMap[A, B]: Conversion[(A, B), Map[A, B]] =
2    (tuple: (A, B)) => Map(tuple)
3
4  given elemToList[A <: ResourceArgs]: Conversion[A, List[A]] =
5    (elem: A) => List(elem)
```

Finally we have all the builders' methods that we encountered during our visit of the files with the JavaParser project:

```
...
def egressOnlyGatewayId(param: String | Output[String])(using b
    : egressOnlyGatewayIdOwners): Unit =
  b match
    case builder: RouteTableRouteArgs.Builder =>
      param match
        case x: String => builder.egressOnlyGatewayId(x)
        case x: Output[String] => builder.egressOnlyGatewayId(x
            )


def ipv6IpamPoolId(param: String | Output[String])(using b:
    ipv6IpamPoolIdOwners): Unit =
  b match
    case builder: VpcArgs.Builder =>
      param match
        case x: String => builder.ipv6IpamPoolId(x)
        case x: Output[String] => builder.ipv6IpamPoolId(x)


def cidrBlock(param: String | Output[String])(using b:
    cidrBlockOwners): Unit =
  b match
    case builder: RouteTableRouteArgs.Builder =>
      param match
        case x: String => builder.cidrBlock(x)
        case x: Output[String] => builder.cidrBlock(x)
    case builder: SubnetArgs.Builder =>
      param match
        case x: String => builder.cidrBlock(x)
        case x: Output[String] => builder.cidrBlock(x)
    case builder: VpcArgs.Builder =>
      param match
        case x: String => builder.cidrBlock(x)
        case x: Output[String] => builder.cidrBlock(x)
...
```

PulumiUtilFunctionsForScala Also here we start with some default imports, then we have the baseOpts function and then all the constructors for the various resources:

```
1   def baseOpts(using o: CustomResourceOptions.Builder) : Unit =
        {}
2
3   def ami(param: String) (init: AmiArgs.Builder ?=> Unit,
4       initOpt: (CustomResourceOptions.Builder ?=> Unit) =
            baseOpts): Ami =
5     given b = com.pulumi.aws.ec2.AmiArgs.builder()
6     init
7     given bo = CustomResourceOptions.builder()
8     initOpt
9     new Ami(param, b.build(), bo.build())
10
11  def amiCopy(param: String)(init: AmiCopyArgs.Builder ?=> Unit
        ,
12      initOpt: (CustomResourceOptions.Builder ?=> Unit) =
            baseOpts): AmiCopy =
13    given b = AmiCopyArgs.builder()
14    init
15    given bo = CustomResourceOptions.builder()
16    initOpt
17    new AmiCopy(param, b.build(), bo.build())
18
19  def amiFromInstance(param: String)(init: AmiFromInstanceArgs.
        Builder ?=> Unit,
20      initOpt: (CustomResourceOptions.Builder ?=> Unit) =
            baseOpts): AmiFromInstance =
21    given b = AmiFromInstanceArgs.builder()
22    init
23    given bo = CustomResourceOptions.builder()
24    initOpt
25    new AmiFromInstance(param, b.build(), bo.build())
```

Final observations on the generated code The builders' methods are really similar to each other, but they are not all following the same exact pattern across the whol AWS EC2 module. Due to this fact, the generation of the code was affordable for the resources used in the case study, but to cover some corner cases for the rest of the resources some extra work both for the parser and for the algorithm that fills in the template would have been required. Because of lack of time and since was not the final aim of the thesis to develop a complete support of Scala for all the AWS EC2 module, only the partial support for the used resources has been generated.
The constructors, instead, are all similar to each other and a complete support for AWS EC2 has been generated.

Chapter 6

Comparisons of the solutions

Here all the advantages and disadvantages detected while using Typecript and Scala for our case study will be presented

6.1 Code readability in TypeScript vs in Scala

6.1.1 Readability in TypeScript

TypeScript's programs have a really readable code. In the implementation we can appreciate how the passing of the builders methods' parameters have been achieved in a concise way using the JSON format. This is fitting very well in a declarative approach, granting compact code solutions. On the other hand, some lacks of the language caused the code to lose part of the readability and conciseness.

Less readable code with pulumi.all and .apply

Lets consider once again the TypeScript version of the function that is responsible for creating the subnets across the various availability zones:

```
1   protected createAZsSubnets(isPvt: Boolean) : Output<Subnet[]>{
2     this.availableZones = aws.getAvailabilityZonesOutput()
3     return pulumi.all([this.availableZones.names, this.vpc!.id]).apply
          (([azNames, vpcId]) => {
4       let i = 0
5       let listToPushInto: Subnet[] = Array<aws.ec2.Subnet>()
6       azNames.forEach(azName => {
7         let fullName = azName + (isPvt ? "-pvt" : "-pub") + "-subnet-
              typescript"
8         listToPushInto.push(new Subnet(fullName, {
9           vpcId: vpcId,
10          availabilityZone: azName,
11          cidrBlock: isPvt ? this.pvtSubnetsCidrs[i] : this.
                pubSubnetsCidrs[i],
12          tags: {
13            Name: fullName,
14          },
15        },{
16          parent: this.vpc
17        }));
18        i++;
19      });
20      return listToPushInto
21    });
22  }
```

When talking about readability, the `pulumi.all` and the `apply` functions are quite cryptic. It takes us a little time to understand what are the return types of the `pulumi.all` and of the `apply` functions. I remind here that the types of these two functions are fully explained in the The apply's lambda paragraph.
We'll make further considerations on this function in the Readability of the Scala's for-yield vs pulumi.all and .apply paragraph.

6.1.2 Readability in Scala

Scala is in general more verbose than TypeScript, but the extreme flexibility of the language let us define a powerful *syntactic sugar* that allowed for a even more readable and concise solution compared to the TypeScript one. The combination of the various features, among which we can find currying, `using` and `given` keywords, and the monads, grant Scala the possibility to define internal DSLs. Thanks to this characteristic, also in Scala I achieved a surprisingly readable code while defining the resources to be created on our Pulumi *stack*.

Function currying

The currying of Scala used to define our *sugared* functions granted us the possibility to declare the resources with this syntax:

```
val resourceName = sugaredResourceConstructor("res-name") {
  firstParameter(...)
  secondParameter(...)
  ...
```

```
}
```

This code is really readable (it resembles to a function definition) and is perfectly fitting in a declarative approach since we can have a straight list of the parameters we want to set for our resource.

Hidden builders

Thanks to the `given` and `using` keywords, builders aren't manually instantiated and we don't require to explicitly say on which builder instance we are calling the builders' methods. This is letting us have an even more lightweight code that is really just focusing on what we need to instantiate instead of the how we could instantiate it. Let's consider the VPC creation in our Scala solution and how would it is it created instead in a Java solution.
Scala solution:

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

Java solution:

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

Even if our *syntactic sugar* is using the very same APIs used from the Java's solution, the readability and the conciseness are entirely on another level.

Implicit conversion functions to get rid of Map and List constructors while passing a single value

The implicit conversion functions presented in the Implicit conversion functions paragraph let us get rid of the constructor of the Map and of the List if we're interested in passing a single value.
It is common, while declaring a new resource, to pass a single parameter to a builder's method that is actually expecting a list or a map of parameters. In such cases syntax at line 3 of the following block of code could be annoying:

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

But thanks to the implicit conversion functions the final result, as we have seen, is the

desired one:

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

For a single parameter the difference of effort in explicitly writing the map constructor can be negligible, but when it comes to define many resources that use multiple methods that accept collections as input parameters, such a feature can really save us a lot of keystrokes, while keeping our code more readable and simple.

Readability of the Scala's for yield vs pulumi.all and .apply

Lets consider the function that creates the subnets. With respect to the TypeScript solution, the Scala one is more readable:

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

All the logic to unbox the `Output` values in order to execute some logic on them is neatly hidden behind the transparent monadic implementation of the `Output` type. This enables a more elegant and concise solution based just on well known constructs such as the for loop and the map function. This allowed to get rid of the cryptic `pulumi.all` function and let us successfully hid the `apply` function in the monadic implementation, without requiring the user to explicitly use it.
TypeScript needed for us to rely on the `pulumi.all` function in order to craft a single `Output` value out of two, so that the `apply` function could be used to get the work done.
Here instead we are separately unboxing the `Output` values with the combination of the expressiveness of the Scala for yield's enumerators and the "unboxing" operation granted by the `Output[Monad]`.
Furthermore, TypeScript implementation requires us to explicitly insert the generated subnets in a variable as a side effect of the `foreach`. Always in TypeScript, we also have to explicitly return the list at the end of the `apply`'s lambda.

All this is making the TypeScript solution more cumbersome with respect to the clean Scala solution.

6.2 Expressiveness in TypeScript vs in Scala

Are here better expressed the observations made during the Readability of the Scala's for yield vs pulumi.all and .apply paragraph.

6.2.1 pulumi.all and .apply vs for comprehension and monads

TypeScript compensate for its lack of expressiveness with the pulumi.all function

In the pulumi.all paragraph of the subnets creation section, we have seen how the concatenation of `apply` to the `pulumi.all` function let us create the subnets across the various availability zones. Anyway, The process of wrapping two different `Ouptut` values in a single `Output` using the `pulumi.all` function, and then extract that newly created value form the `Output` *context* is verbose and complex.
The `pulumi.all` function is effective but its existence is required to compensate for the lack of expressiveness of the TypeScript language.

Scala's expressiveness let us get rid of pulumi.all

The functional nature of the Scala language let us get rid of the `pulumi.all` function thanks to the monad implementation for the `Output` type. The for yield, exploting the monadic type of `Output`, is letting us unbox the single `Output`s separately, in a flexible and intuitive way.
Furthermore, the `apply` function that we had to explicitly invoke in the TypeScript solution, became in Scala the actual implementation of the `flatMap` function for the `Output[Monad]`, being totally invisible to the user. This achievement is perfectly depicting how the expressiveness of Scala permitted us to satisfy our need of applying functions on multiple separated `Output` values without resorting to an ad-hoc function created only for that specific purpose (`pulumi.all`). In fact, we would be able to implement a monad for any *context* type, while `pulumi.all` is working exclusively with the `Output` type. Moreover, the for loop is is much more intuitive to an user rather than the `pulumi.all` and the explicit `apply` functions combination.

Chapter 7

Conclusions

7.1 The potential of Scala for Pulumi

The infrastructure scenario kept evolving in the last decades allowing for more and more flexible solutions. By the way this flexibility didn't come for free. The complexity and amount of configuration required grew along with the increasing flexibility. The tools for the management of the configuration evolved as well, reaching their peak with Pulumi. It is capable to offer a single solution to manage all the desired cloud resources offered by many cloud providers. This feature meets the requirements of many companies that, due to the cloud providers competition, often opt for a multi-cloud solution of their infrastructure. Moreover, Pulumi supports many general purpose programming languages instead of relying on less expressive markup languages. With this feature, Pulumi can keep up with the always more demanding requisites posed from the configuration management increasing complexity.
Though, some languages are more expressive than others, and sometimes they are carrying over some extra complementary benefits or shortcomings. We have already seen how TypeScript's limited expressiveness had us resort on the `pulumi.all` function and we already discussed how the building system of TypeScript is not as mature as the java one, how the duck typing represents a limitation for the refactoring tools offered from the IDEs, and more. Scala, on top of having more expressive constructs that allowed for a better expressiveness, is also carrying over many complementary benefits, such as: many libraries, interoperability with the other JVM languages, a powerful building system, great IDEs to support the refactoring of the code, and a better management of the logical structure of the code as packets.
TypeScript's high readability is a really appreciated feature when it comes to define resources with Pulumi. On the other hand Scala, by nature, is more verbose. Anyway, the functional paradigm of Scala and its nature prone to the definition of internal DSLs, granted me the possibility to define a powerful *syntactic sugar*. Such *sugared* functions let me achieve a very readable solution that, in some cases, was even more concise with respect to the TypeScript's one.
So all the heavy machinery that Scala was bringing over turned out to be the solution to achieve a both robust, concise, and readable solution.
The main successes of the work are mainly two. First, it has been shown that Scala is offering the chance to achieve very readable and concise IaC solutions while not giving up on solid complementary tools and benefits. Second, we proved how the support of a

such an interesting language can be added to Pulumi exploiting the already supported Java APIs of Pulumi and the interoperability between the JVM languages (like Scala and Java), greatly reducing the effort needed to develop a Scala support for Pulumi.

7.2 Future improvements

The work done with my thesis was able to analyze only some of the many challenges that Pulumi has to offer to the various programming languages while defining a solution for a given use case. Yet, the results obtained wanted to show that the flexibility of Scala can potentially address any kind of difficulty encountered. Therefore, a further extension of the work done with the aim to investigate which other supplementary possibilities Scala has to offer with Pulumi would be indeed interesting.
If possible, an official and complete support of Scala for Pulumi would be, in my opinion, a fantastic addition to the roster of languages of Pulumi.

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.

Medium - Macros 3 without pain. URL: https://medium.com/codex/scala-3-macros-without-pain-ce54d116880a.

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

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

Pulumi architecture. URL: https://www.pulumi.com/docs/intro/concepts/how-pulumi-works/.

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/.