POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Master Degree Thesis

Development, Test and Application of a framework for cloud serverless services

Thesis Supervisor

Dr. Ing. Boyang Du

Candidate

Andrea SANTU

matricola: 251579

Intership Tutor

Dott. Antonio Giordano

ACADEMIC YEAR 2020-2021

Abstract

todo

brief description of the thesis

Acknowledgements

todo

Contents

Chapter 1

Introduction

1.1 Cloud computing models

Between the various types of cloud computing architectures, in the last few years have emerged three main models, through which to develop web applications. These are IaaS, PaaS e SaaS. Each of the models is characterized by an increasing level of abstraction regarding the underlying infrastructure.
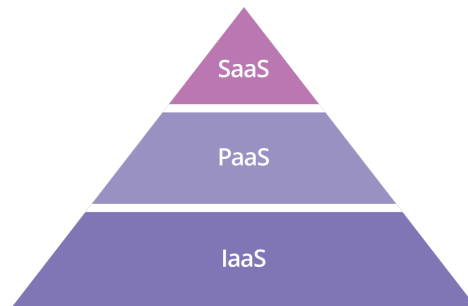

Figure 1.1. IaaS, PaaS, SaaS pyramid

Infrastructure as a Service (IaaS) Infrastructure refers to the computers and servers than run code and store data. A vendor hosts the infrastructure in data centers, referred to as the cloud, while customers access it over the Internet. This

eliminates the need for customers to own and manage the physical infrastructure, so they can build and host web applications, store data or perform any kind of computing with a lot more flexibility. An advantage of this approach is scalability, as customers can add new servers on demand, every time the business needs to scale up, and the same apply also if the resources are not needed anymore. Essentially servers purchasing, installing, maintenance and updating operations are outsourced to the cloud provider, so customers can spend fewer resources on that and focus more on business operations, thus leading to a faster time to market. The main drawback of this approach is the cost effectiveness, as businesses needs to over-purchase resources to handle usage spikes, this leads to wasted resources.

Platform as a Service (PaaS) This model simplify web development, from a developer perspective, as they can rely on the cloud provider for a series of services, which are vendor dependent. However some of them can be defined as core PaaS services, and those are: development tools, middleware, operating systems, database management, and infrastructure. PaaS can be accessed over any internet connection, so developers can work on the application from anywhere in the world and build it completely on the browser. This kind of simplification comes at the cost of less control over the development environment.

Software as a Service (SaaS) In this model the abstraction from the under-lying infrastructure is maximized. The vendor makes available a fully built cloud application to customers, through a subscription contract, so rather than purchasing the resource once there is a periodic fee. The main advantages of this model are: access from anywhere, no need for updates or installations, scalability, as it's managed by the SaaS provider, cost savings. However there are also main disadvantages, that makes this solution not suitable in some cases: developers have no control over the vendor software, the business may become dependent on the SaaS provider (vendor lock-in), no direct control over security, this may be an issue
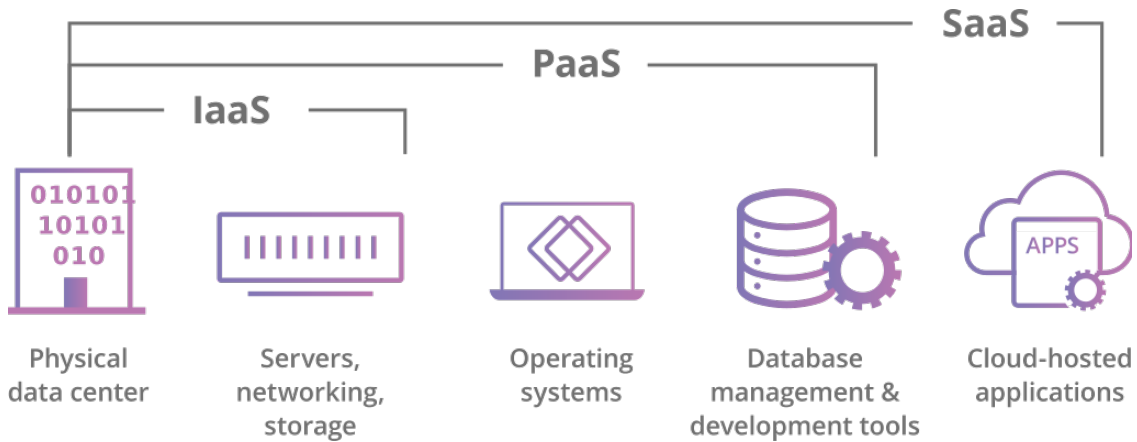
especially for large companies.

Figure 1.2.   IaaS, PaaS, SaaS diagram

Another model has recently been added to the three main cloud computing models, named Backend as a Service (Baas). This model stands, with some differences, at the same level of PaaS, and it's suited especially for web and mobile backend development. As with PaaS, BaaS also makes the underlying server infrastructure transparent from the developer point of view, and also provides the latter with api and sdk that allow the integration of the required backend functionalities. The main functionalities already implemented by BaaS are: database management, cloud storage, user authentication, push notifications, remote updating and hosting. Thanks to these functionalities there may be a greater focus on frontend or mobile development. In conclusion BaaS provides more functionalities with respect to the PaaS model, while the latter provides more flexibility.

1.2   Serverless paradigm

The downsides of the previously described approaches varies from the control on the infrastructure and on the software, to scalability problems, to end with cost and resources utilization effectiveness. With the aim of solving these problems, the

major providers started investing on a new cloud computing model, named Function as a Service (FaaS) and based on the serverless paradigm. Such a paradigm is based on providing backend services on an as-used basis, with the cloud provider allowing to develop and deploy small piece of code without the developer having to deal with the underlying infrastructure. So despite the terminology, serverless does not means without servers, as they are of course still required, but they are transparent to developers, which can focus on smaller pieces of code. With this model, rather than over purchase the resources, to ensure correct functionality in all workload situations, as happens in the IaaS model, the vendor charges for the actual usage, as the service is auto-scaling. Thanks to this approach consumer costs will be fine grained as shown in 1.3.

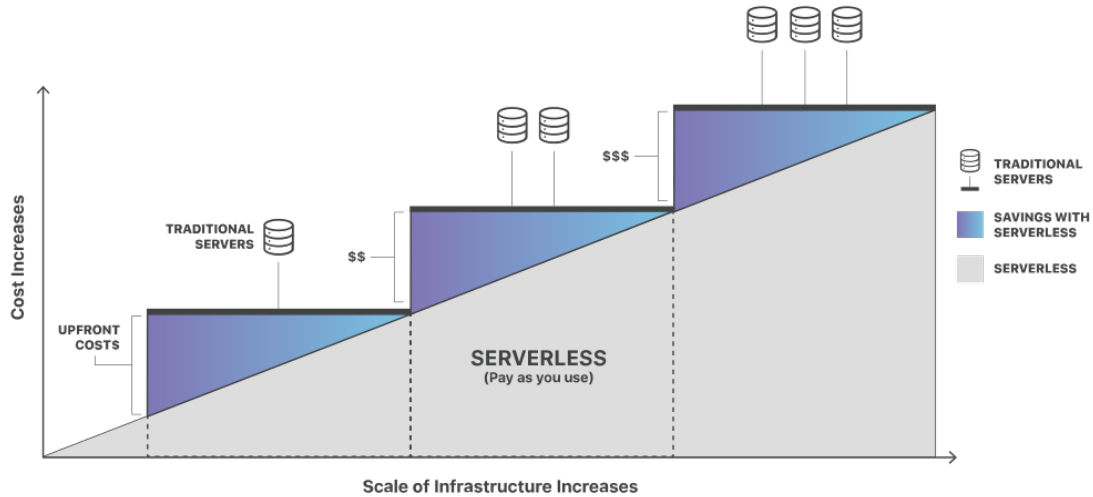Figure 1.3. Cost Benefits of Serverless

Being the underlying infrastructure transparent for the developer, you get the advantage of a simpler software development process, and this advantage characterize also the PaaS model. Furthermore being the service auto-scaling, is possible to obtain a virtually unlimited scaling capacity, as it happens in the IaaS model, where the limit is the cloud provider availability.

An implementation of the serverless paradigm is the cloud model named Function as a Service (FaaS), which allows developers to write and update pieces of code on the fly, typically a single function. Such code is then executed in response to an event, usually an api call, but other options are possible, so it executed regardless of the events, and this lead to the previously described benefit regarding scalability and cost effectiveness. Furthermore, through this model turns out to be more efficient to implement web applications using the modular approach of the micro services architecture (1.4), since the code is organized as a set of independent functions from the beginning.
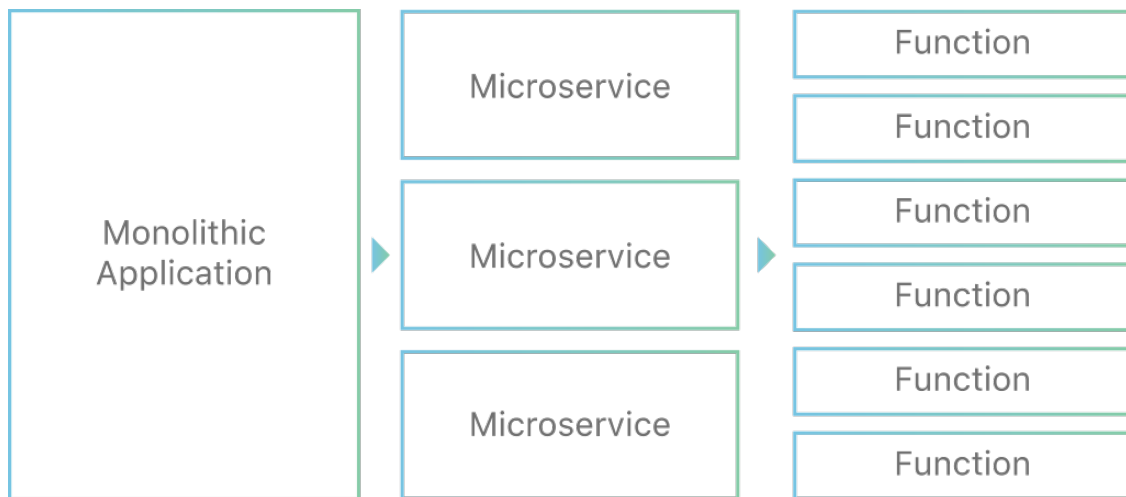
Figure 1.4.   Monolithic to Micro services application

So the main advantages of the FaaS model are: improved developer velocity, built-in scalability and cost efficiency. As each approach, there are also drawbacks, in this case developers have less control on the system, and an increased complexity when it comes to test the application in a local environment.

The first cloud provider to move into the FaaS director has been Amazon, with the introduction of aws lambda in 2014, followed by microsoft and google, with azure function and cloud function respectively in 2016.

1.3 Serverless Framework

Shortly after the release of the service Aws lambda functions, has been introduced, in 2015, the Serverless framework, with the main objective of making development, deploy and troubleshoot serverless applications with the least possible overhead. The framework consists of an open source Command Line Interface and a hosted dashboard, that combined provide developers with serverless application lifecycle management.

Although the serverless framework, given the number of cloud providers supported, aim to be platform agnostic, this document will consider primarily the Aws provider, for both the usage of the serverless framework and the subsequent development of the Restlessness framework. This choice is due to the maturity of the platform with respect to what the competitors. Serverless supports all runtime provided by Aws, corresponding to the most popular programming languages such as: Node.js, Python, Ruby, Java, Go, .Net, and others are on development. This document will focus on the Node.js runtime along with the typescript programming language.

The main work units of the framework, according to the FaaS model, are the functions. Each function is responsible for a single job, and although is possible to perform multiple tasks using a single function, it's not recommended as stated by the design principle Separation of concerns. Each function is executed only when triggered by and Event, there are a lot of events, such as: http api request, scheduled execution and image or file upload. Once the developer has defined the function and the events associated to it, the framework take care of creating the necessary resources on the provider platform.

The framework introduces the concept of Services as unit of organization. Each service has one or more functions associated to it and a web application can then be composed by multiple services. This structure reflects the modular approach of the micro services architecture described previously.

A service is described by a file, located at the root directory of the project, and composed in the format Yaml or Json. Below is a simple serverless.yml file (listing 1.1) , it defines the service users, which contains just a function, responsible of creating a user. The handler field specify the path to the function code, in this case the framework will search for a handler.js file, exporting a usersCreate function, as show on listing 1.2.

Listing 1.1. Simple serverless.yml file

```
service: users
provider:
    name: aws
    runtime: nodejs12.x
functions:
    usersCreate:
        handler: handler.usersCreate
        events:
            - http: post users/create
```

Listing 1.2. Simple handler function

```
async function usersCreate(event, context) {
    const user = {
        name: 'sample_name',
        surname: 'sample_surname'
    }
    await mockDb.createUser(user)
    return user
}
```

Serverless is flexible and does not force a fixed structure of the project, that task is up to the developer. Defined that structure, the service can be deployed using

Figure 1.5.   Simple Serverless project structure

```
./
├── handler.js
├── serverless.yml
```

the Serverless CLI, on the chosen provider, as shown on listing 1.3.

Listing 1.3.   Deploy command

```
$ serverless deploy
```

The deploy command creates the necessary aws resources, in this case they are: a lambda function corresponding to the usersCreate function and an api gateway to handle http requests. It is then possible to test the newly created resource by making requests to the url returned by the CLI, as shown in specifying the resource path /users/create. It is possible to invoke online functions also directly from the CLI, specifying the identifier of the function used in the serverless.yml file, as shown on listing 1.4

Listing 1.4.   Invoke command

```
$ serverless invoke −f usersCreate
```

The development and deploy process shown for a service with a single function remains the same as the service complexity grows, in particular it is possible to modify and deploy a single function at a time, since each function has its own resource associated. This process gets along with the previously described micro services architecture.

1.3.1   Advantages

The main advantages of using the Serverless framework are:

- Provider agnostic: the framework aims to be independent from the chosen cloud provider, thus avoiding vendor lock-in. In practice this feature is not

achieved completely, as the configuration file serverless.yml may be different across providers. However the main structure remains the same, and that simplify providers migration.

- Simplified development: the CLI commands simplify the development process, from the deploy from the testing of the deployed functions.

- Extensible: is possible to develop plugins that integrate with the CLI commands lifecycle, increasing their functionalities.

- Dashboard: the hosted dashboard allow monitoring and tracing of the deployed functions and services.

1.3.2 Disadvantages

The main advantages of using the Serverless framework and the Serverless paradigm are:

- Compilation of the configuration file may become tedious as the project grows.

- The framework is extremely flexible regarding the project structure and that is an advantage, however this can also be a drawback as it's up to the developer to find a suitable structure, and this means less time spent on business related tasks.

- Unit testing: it is possible to test a deployed function easily, however for big projects, where it's necessary to test a lot of functions, this may become cumbersome.

- Resource threshold: for projects created with Aws, a single serverless.yml file may create up to 200 resources, and if exceeded the deploy operation fails. Since each function is responsible for the creation of about 10 resources, is very easy to exceed this limit. The only solution so solve this problem is

to split the functions across multiple services, hence different serverless.yml configuration files.

- Cold start: inherent overhead of the current implementation of the serverless paradigm. Since each function is executed only in response to an event, a certain amount of time is required for resources initialization.

1.4 The idea behind Restlessness

The open source framework named Restlessness was born with the goal of improving the developer experience of the Serverless framework, by addressing its encountered problems. The framework is composed by a Command Line Interface and a frontend application with an associated web server running locally. In particular the main functionalities that the framework aims to provide are:

- Creation of a new project, through the CLI, based on the typescript language and with a standard structure.

- A local Web Interface that allow creating and managing project resources, functions, with their associated events, and models.

- The creation of a standard unit testing structure for each function, and based on the jest library.

- A standard validation structure for function's input, based on the yup library.

- Deploy of multiple services with a single CLI command, to deal with the resource threshold limitation of Aws.

By addressing those points the framework aims to give developers the tools to focus on writing business code rather than spend time on boundary problems, that are important, but there may be the risk of solving the same problems multiple times (reinventing the wheel), which may be avoided.

1.5   Related Works

@TODO

Are there other similar framework? What are the differences? Why use restlessness instead?

1.6   Tools

Several tools were used during the development of the project, varying from the ones supporting code development, to the organizational ones. Below is a list of them:

GitHub   Version control platform based on the Git system. It has been used for the development, organization and management of the main project *Restlessness*, as well as the drafting of this document. Both are available for consultation:

- Restlessness: https://www.github.com/getapper/restlessness

- Thesis: https://www.github.com/androsanta/Thesis

Slack   Business communication platform. It has been used for team communication to achieve a more direct and private interaction with respect to what *GitHub* offers.

CircleCi   Continuous integration platform. It has been used for testing and deploying operations for all restlessness packages.

Serverless   Open source framework that simplify the development and deployment of applications based on the serverless architecture, on the best known compute services.

Aws   Amazon Web Services. Platform for cloud computing services. It has been used for deployment and testing of application created with the *Restlessness* framework.

Node.js   Open source Javascript runtime environment that allow Javascript code execution outside a web browser.

Npm   The *Node.js* package manager. It has been used for project's dependencies and for publishing of all *Restlessness* packages.

Typescript   It extends the Javascript language by adding type definitions, and a transpiler that generates code that runs anywhere Javascript runs, from the browser to *Node.js*. It has been the main programming language used.

WebStorm   Javascript IDE from JetBrains. It has been used for all code development.

Chapter 2

Restlessness

The framework is composed by different components, listed here:

- Command Line Interface: together with the Web Interface, this is the main component with which users interact the most. It is available as the @restlessness/cli package on npm.

- Restlessness frontend: Web Interface with which it is possible to create resources and manage the project. It is part of the CLI.

- Restlessness backend: api service running locally, created with the Restlessness framework itself. It is used by the Web Interface to provides its functionalities.

- Restlessness core: core package of the framework, it contains all the classes and functions that provides the framework functionalities. It is available as @restlessness/core package on npm.

2.1 Project creation

The Restlessness CLI is available for installation on the npm platform, as described on 4. Once installed, the first step toward using the framework is the creation of a

new project, and that is possible using the 'new' command, as shown on (todo ref)
Once the command has finished, a new folder has been created, with a completely
structured restlessness project, as can be see in 2.1.

Figure 2.1.   Sample Restlessness project structure

```
./
├── .restlessness.json
├── configs/
│   ├── authorizers.json
│   ├── daos.json
│   ├── default-headers.json
│   ├── endpoints.json
│   ├── envs.json
│   ├── models.json
│   └── schedules.json
├── envs/
│   ├── .env.locale
│   ├── .env.production
│   ├── .env.staging
│   └── .env.test
├── serverless-services/
│   ├── offline.json
│   └── shared.json
└── src/
    ├── exporter.ts
    └── schedulesExporter.ts
```

The sample project shown in 2.1 however, does not include all generated files,
as some of them are not strictly part of the framework, but are required from other
used tools, in particular:

- .eslintrc.json: configuration file of the linter eslint.

- .gitignore: list intentionally ignored files from the git tracking system.

- package.json: entry point of every npm project, it lists the project dependen-
  cies, as well as other project related information, such as the project name and
  version.

18

- package-lock.json: npm generated file, contains a snapshot of the version of all dependencies, with the goal of obtaining reproducible builds.

- tsconfig.json: configuration file for the Typescript compiler.

The first noticable difference with respect to a plain serverless project is the lack of a serverless.yml (or serverless.json) file under the root. In fact, due to the resource threshold limitation imposed by Aws, has been decided to let the framework manage the presence of multiple services inside a single project, so setting up a structure that is standard and micro services oriented. Following this structure, all serverless.yml file correspondents to the various services, are located under the serverless-services folder. Has been decided to format those configuration files using Json, instead of Yaml, to simplify their handling and modification by the framework, given that Typescript handle Json files and objects natively. After creation, the sample Restlessness project already contains two services, named shared and offline, and they are required for the framework to work.

The shared service will contains all shared resources that can be used by all the other services. This is the case for the api gateway (@todo ref aws api gateway description) because, since it is responsible for handling http requests, it is convenient to have a single url for all services, rather than one for each service. Other shared resources may be simple functions or authorizers. The offline service is required to handle local development, as it contains the resource definition of all services.

Other created files are: configuration files, under the config folder, environment files, containing environment variables for different deployments, source code, under the src folder, and a .restlessness.json file, used to store project related information needed by the framework.

2.2 Local development

Lo sviluppo in locale richiede la presenza di diversi processi, come detto prima, lato framework sono necessari, una web interface ed un servizio api, mentre lato progetto, è necessario poter testare l'applicazione che si sta sviluppando. La cli si occupa di gestire questi 3 processi attraverso un singolo comando, denominato dev (todo ref). In particolare sia il processo del progetto, sia il processo di Restlessness backend vengono eseguiti utilizzando il plugin serverless-offline, il quale consente di emulare un api gateway, creando cosi un http server locale. Riguardo il frontend, viene invece utilizzato il pacchetto npm 'serve', attraverso il quale è possibile creare un http server per servire file statici. Tale comando inoltre si occupa di far partire tali processi seguendo the dependency order, ovvero: rln backend, rln frontend ed infine the project's process.

Un altro compito del comando dev è inoltre quello di implementare un inter process communication tra se stesso ed il processo backend. Questo è necessario in quanto quando vengono create delle risorse, come ad esempio endpoints o schedules, i corrispondenti file devono essere compilati da Typescript, ed inoltre il plugin serverless-offline deve essere fatto ripartire perchè le nuove risorse siano disponibili tramite l'http server.

Come mostrato in (@todo ref dev command execution) il comando dev riceve in input il nome di un environment, in quanto si occupa di caricare le variabili d'ambiente dal corrispondente file dalla cartella .envs (2.7).

2.3 Resource creation

L'interfaccia web si presenta come in figura (@todo ref), e fornisce alcuni dettagli riguardo il progetto, come l'organizzazione della dashboard serverless, l'applicazione, alla quale fanno parte i vari servizi creati (1.3), ed infine la regione del data center sul quale l'applicazione sarà deployata. Sono poi presenti diverse shortcuts tramite

le quali è possibile creare e visionare le risorse, quali: endpoints, schedules, services and models.

La creazione di un servizio viene gestita internamente dalla classe JsonServices, la quale fornisce le funzionalità CRUD per i configuration files dei vari servizi definiti. La creazione di un nuovo servizio (e.g users) corrisponderà alla creazione di un nuovo file di configurazione (e.g users.json).

Essendo Restlessness un framework per i servizi serverless la risorsa principale che può essere definita sono le funzioni, e al momento è possibile definire due tipi di funzioni, a seconda dell'evento che le triggera. Esse sono gli endpoints, legati a degli eventi http, e gli schedules, legati a degli eventi programmati, come ad esempio i cron.
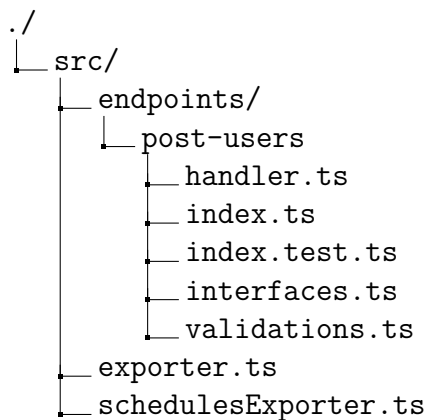
2.3.1 Endpoints

Dalla web interface è possibile creare un endpoint specificando i seguenti campi, come mostrato in (@todo ref):

- Service: il servizio a cui la funzione deve appartenere.

- Route: the path corresponding to the serverless function.

- Method: http method.

- Authorizer: this optional field sets a further function, that perform the authorization operation, granting or denying access to the specified function, as explained on (@todo ref).

Durante la creazione di un endpoint, il framework si occupa di salvare le informazioni fornite nei file di configurazione, in particolare nel file config/endpoints.json, e di creare dei code template per lo sviluppo della funzione corrispondente. Come mostrato in figura 2.2, viene creata una cartella sotto src/endpoints, all'interno della quale sono stati generati diversi files.

Figure 2.2. Structure of a new endpoint folder

```
./
└── src/
    ├── endpoints/
    │   └── post-users
    │       ├── handler.ts
    │       ├── index.ts
    │       ├── index.test.ts
    │       ├── interfaces.ts
    │       └── validations.ts
    ├── exporter.ts
    └── schedulesExporter.ts
```

The developer can then code the function in handler.ts, which already contains a template (@todo ref) and define the validation object in validations.ts (@todo ref). It is also possible to exploit the Typescript functionalities, defining the various interface for the request, response and query parameters objects, all under the interfaces.ts file (@todo ref). The actual function entry point that will be executed once deployed is defined in the file index.ts (@todo ref). This function is created binding the function LambdaHandler input with the handler function and validation object. LambdaHandler is a core function of the framework, its purpose is to parse the request payload and or query parameters, load the environment variables (@todo ref) and execute the lifecycle hooks of the installed addons (@todo ref). After those operation the LambdaHandler execute the actual handler function.

2.3.2 Schedules

Schedules are serverless functions that are triggered by a programmed event, such as a cron job or a rate event, an event that is fired up periodically, based on the time interval provided. By creating a Schedule from the Web Interface the framework creates the necessary template files under src/schedules as shown on 2.3, and also save the provided information under the config/schedules.json file.

22

Figure 2.3.   Structure of a schedule endpoint folder

```
./
└── src/
    └── schedules/
        └── clean/
            ├── handler.ts
            ├── index.ts
```
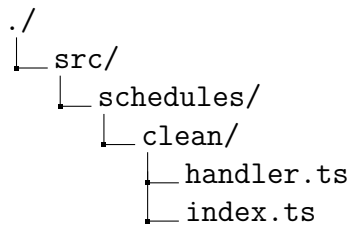
The structure of the template files is similar to the one generated for endpoints, but simpler. The handler.ts file contains the function that the developer has to code, while the index.ts file is the entry point. The core function ScheduleHandler is used to wrap the handler function, the same way as happens for endopints, with the purpose of executing the framework lifecycle hooks (@todo ref).

2.3.3   Models

@todo

write here or under extensions when talking about data access objects?

2.4   Test

A test template is also provided when creating a new endpoint (@todo ref), and it is based on the popular testing library jest. In addition to the jest library, Restlessness provides also a TestHandler class, which makes testing the endpoint straightforward. Inside the beforeAll function it performs initialization operations, such as loading the correct environment variables, while the function invokeLambda executes the endpoint function providing automatically the event and context objects (@todo ref), simulating this way an http event. The fact that serverless is based on function makes possible using a simple testing structure as the one presented, as it's not necessary for example to actually starts an http server to test the endopints.

2.5 Api documentation

@todo swagger generated documentation

2.6 Deploy

The Serverless Framework already provides a command for the deploy operation, as shown on 1.3, however with the micro services oriented structure suggested by Restlessness this operation is more elaborate, as it involves the deploy of more than one service, in a particular order. This is necessary because of the presence of the shared resources service, so to successfully deploy a service that uses resources from the shared one, it is necessary that those resources already exists. The correct deploy ordering is then shared service first, followed by all the other services. It should be noted that the offline service is not involved in the deploy process as it's used only for local development. To address this operations the Restlessness CLI provides a custom deploy command (@todo ref), and a complementary remove command that removes all the services enforcing an opposite ordering.

It is possible to deploy the application on different environments, as shown on (@todo ref), otherwise the command assume staging as the default environment. It is also possible to perform the deploy of just a single service, to keep the whole development, test and deploy process fast and easy, when making small changes, in accordance with the serverless paradigm.

Siccome l'operazione di deploy involves più servizi, è importante che le informazioni tra essi siano consistenti, specie al momento del deploy. Per questo motivo, under the hood, il comando deploy si occupa anche di effettuare questo controllo, denominato healthCheck. In particolare viene controllato che i vari servizi appartengano alla stessa applicazion, organizzazione ed aws region, e che non esistano servizi con funzioni associate allo stesso path. Quest'ultimo punto è dovuto al fatto che venga utilizzato uno shared api gateway.

2.7 Environment variables

An important aspect when developing web applications is the handling of different deploying environments, as each one of them requires different configurations, mostly for sensitive informations, such as database credentials. Has been decided to handle this information with different environment files, storing environment variables. At project initialization the framework creates 4 different evironments: locale, test, staging and production. Each environment has an associated type and stage (@todo ref). The type represent the purpose of that environment, below are the available types:

- test: environments used only for testing, which can happen locally but also through CI platform.

- dev: environments used for local development

- deploy: environments that can be deployed

All information about the environments (name, type, stage) are stored in the configuration file config/envs.json and are managed by the JsonEnvs class.

Environment variables are stored in the format key=value and variable expansions is supported, so the value of a key can be another variable, using the syntax (@todo ref). Each environment is then stored under the envs/ directory, in the form .env.<name>, and the interaction with those files is handled by the EnvFile class. The load and expansion operation is performed differently depending on the operation, local development or deploy. During local development it is the dev command that load the environment specified in input (2.2). During deploy instead, the environment file is expanded and copied under the project root, in a file named .env, as this makes deploying from CI straightforward. Then at runtime the .env is automatically loaded by the LambdaHandler or ScheduleHandler functions (2.3.1, 2.3.2).

Chapter 3

Restlessness Extensions

The framework has been designed from the beginning with the possibility of extending its funcionalities using external packages. To achieve this, has been defined an AddOnPackage class, containings the following lifecycle hooks:

- postInstall: executed after the addon package has been installed. Here it's possible to perform initializiation operations.

- postEnvCreated: executed after a new environment has been created, so the addon can add its own environment variables if needed.

- beforeEndpoint: executed before the corresponding function of an endpoint. Here it's possible to perform resource initializiation, for example opening a database connection.

- beforeSchedule: as for endpoints, it's executed before the corresponding function of a schedule.

In addition to this class Restlessness provides also more specific classes, for authentication and data access.

3.1 Authentication

auth-jwt and auth-cognito packages

3.1.1 Usage example

@todo

3.2 Data Access Object

To simplify the creation of a Data Access Object, Restlessness provides the abstract class DaoPackage (@todo ref code), which extends the AddOnPackage class previously defined. In addition to the previously defiend hooks, classes implementing DaoPackage, should implement also the modelTemplate method, and a base dao class, to which we will refer to as DaoBase. This latter class should provides the main Dao functionalities, while the code template returned by modelTemplate should define a class that extends the DaoBase one.

Restlessness already provides a Dao package for the popular non relational database mongodb, and it's available on the npm platform as '@restlessness/dao-mongo'. That package exports two main components, an implementation of the DaoPackage class, and a MongoBase class, the DaoBase class containings the main Dao functionalities for CRUD operations, as shown on (@todo ref code).

Users of the package can then create models based on the MongoBase class through the Restlessness Web Interface (2.3.3). The creation of that model is made possible by implementing the DaoPackage.modelTemplate method, as shown on (@todo ref model template)

28

3.2.1 Database Proxy

todo

show data about number of connections until crash in normal situation and when using the proxy

3.2.2 Usage example

@todo

Chapter 4

Development

4.1 Github

Useful tools provided by GitHub (projects to handle tasks) Roadmap, development process/flow (issues, pull requests...)

4.2 Continuous Integration

circle-ci

parlare anche di npm (insieme a ci)

Chapter 5

Application

Application to real projects with emerged problems

5.1 FGA covid school api

brief description of the project

Problems arised by using restlessness (mainly the need for a database proxy and a solution for cold start, i.e. warmup plugin)

5.2 Gbsweb Claranominis api

brief description of the project

Problems arised by using restlessness (mainly necessary migration to micro-services structure)

rimuovere claranominis e parlare delle 2 problematiche solo per Spazio alla scuola parlare anche dell'attenzione mediatica e del numero di utenti (per mostrare le capacità di scalabilità di sls)

Chapter 6

Deployment

6.1   Aws

todo

detailed description of what resources are created when deploying

6.2   Why Aws

also talk about the various stage, environment handling

Chapter 7

Conclusions

todo?? cosa si potrebbe fare per migliorare il framework -> integrazioni con altri db
cross platform in futuro? includere gli altri cloud providers per renderlo agnostico
testing -> non c'è ancora nulla per gli integration test ricavare info dalla dashboard,
   portare alcune funzioni della cli sulla web interface e viceversa