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. Magistrale Antonio Giordano

Academic year 2020-2021

Abstract

todo

brief description of the thesis

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.

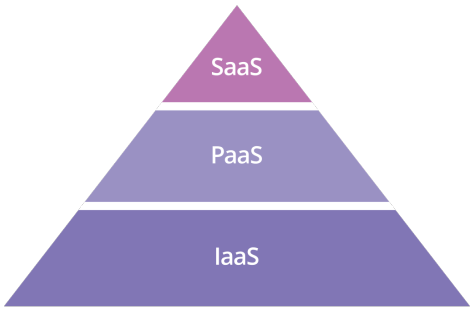


Figure 1.1. IaaS, PaaS, SaaS pyramid

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 underlying 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.

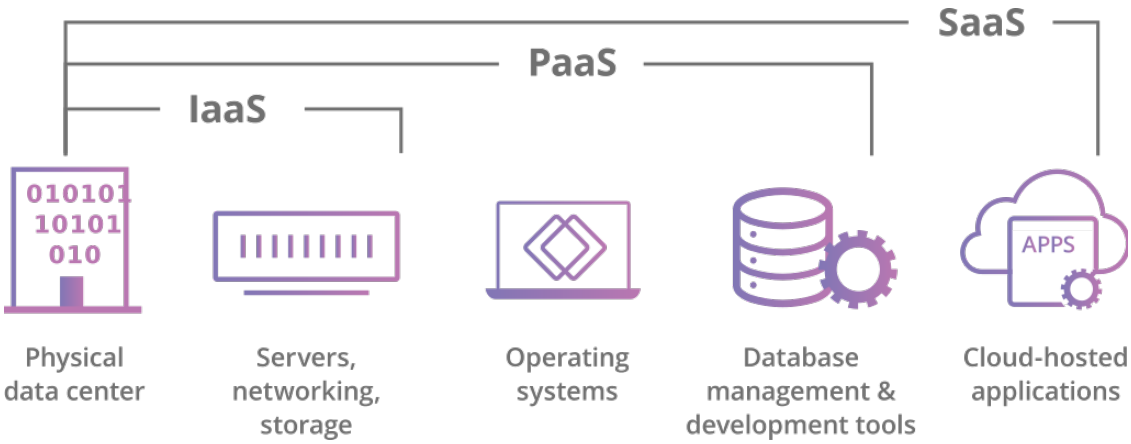


Figure 1.2. IaaS, PaaS, SaaS diagram

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.

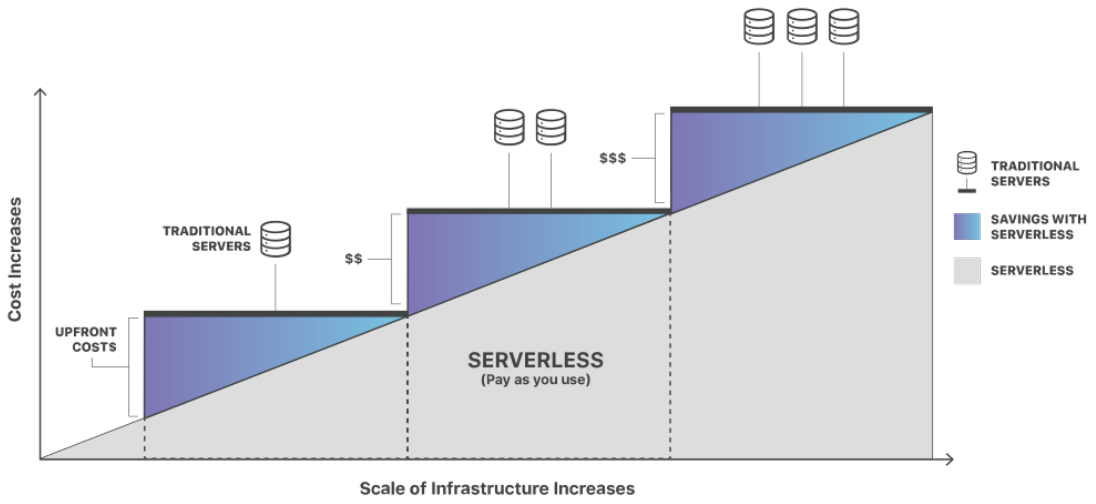


Figure 1.3. Cost Benefits of Serverless

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.

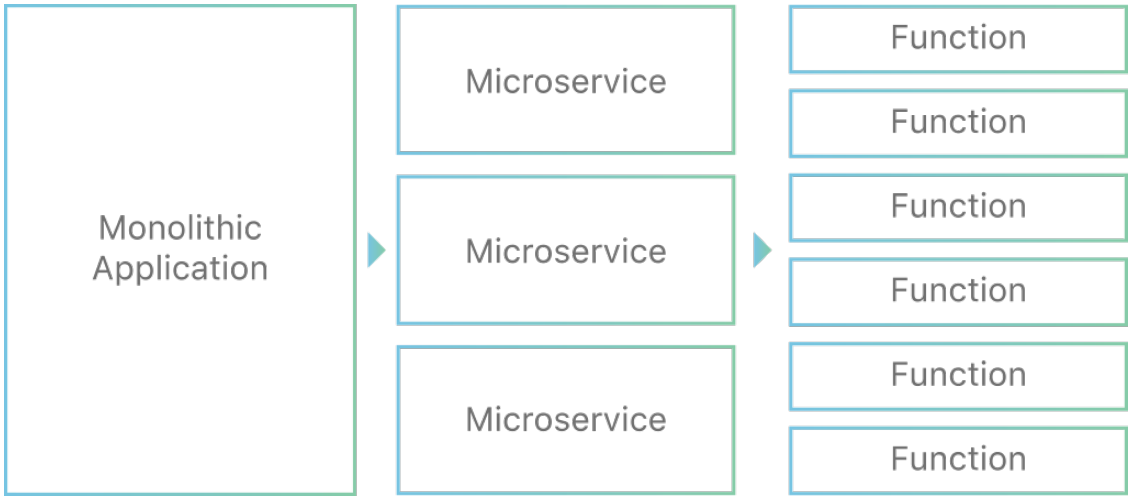


Figure 1.4. Monolithic to Micro services application

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

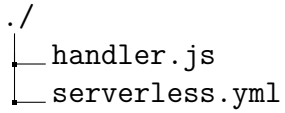
```
service: users
provider:
    name: aws
    runtime: nodejs12.x
functions:
    usersCreate:
        handler: handler.usersCreate
        events:
            - http: post users/create
```

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



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 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

Figure 2.1. The Restlessness logo

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 chapter 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 listing 2.1.

Listing 2.1. New command

```
$ restlessness new rln_project
```

Once the command has finished, a new folder has been created, with a completely structured restlessness project, as can be see in figure 2.2.

The sample project shown in figure 2.2 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 dependencies, as well as other project related information, such as the project name and version.
- 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 noticeable 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

Figure 2.2. 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
```

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

The local development requires the presence of different processes, and as previously said, framework's side are necessary a web interface and an api service, and also a the project's process to be able to test it. The CLI handles those 3 processes through a single process, named dev (listing 2.2). In particular, both the project's process and Restlessness backend, are executed using the Serverless plugin serverless-offline, which allow simulating an api gateway, effectively creating a local http server. Instead for the frontend process has been used the npm package serve, through which is possible to create an http server that serve static files. Furthermore the dev command takes care of executing those processes following the dependency order, which is: Restlessness backend, frontend and finally the project's process.

Another task of the dev command is to implement inter process communication between itself and the backend process. This is necessary as when resources are created, for example endpoints or schedules, the corresponding files need to be compiled by typescript and also the serverless-offline plugin needs to be restarted for those resources to be available from the http server.

As shown on listing 2.2, the command receives the environment name in input, as it takes care of loading the corresponding environment variables from the folder .envs, as explained on section 2.7.

Listing 2.2. Dev command

```
$ restlessness dev locale
```

2.3 Resource creation

The Web Interface looks like in the figure 2.3, and provides some project details, such as serverless organization, application (section 1.3), and finally the aws data center region to which the project will be deployed. The main functionalities are then available through some shortcuts, that allow creating and consulting resources, such as endpoints, schedules, services and models.

Figure 2.3. Restlessness Web Interface

Being Restlessness a framework for serverless services, the primary resource that can be defined are functions, and at the moment it is possible to define two type of functions, based on the event that triggers them. They are endpoints, for http event, and schedules, for programmed events, such as cron jobs.

2.3.1 Endpoints

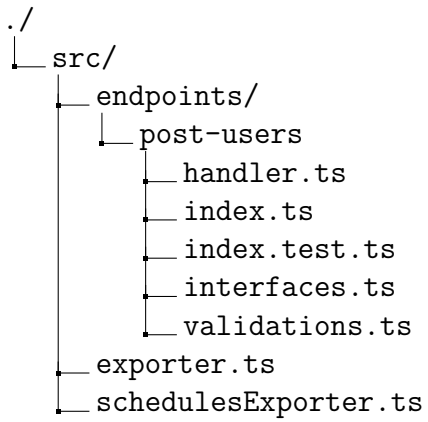
It is possible to create an endpoint from the Web Interface, by specifying the following fields, as shown on figure 2.4:

- Service: the service to which the function must be associated.
- Route: the path corresponding to the serverless function.
- Method: the http method.
- Authorizer: this optional field sets a further function, that perform the authorization operation, granting or denying access to the specified function.

Figure 2.4. Creation of and endpoint

During the endpoint creation, the framework takes care of saving the provided information on the configuration file config/endpoints.json, and to create code template for the development of the corresponding function. As shown on figure 2.5, has been created a folder under src/endpoints, using the notation http method plus normalized value of the http path.

Figure 2.5. Structure of a new endpoint folder



The developer can then code the function in handler.ts, which already contains a template (listing 2.3) and define the validation object in validations.ts (listing 2.5). 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 (listing 2.6). The actual function entry point that will be executed once deployed is defined in the file index.ts (listing 2.4). 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 (section 2.7) and execute the lifecycle hooks of the installed addons (chapter 3). After those operation the LambdaHandler execute the actual handler function.

Listing 2.3. handler.ts content

```
export default async (req: Request) => {
    try {
        const {
            validationResult,
            payload,
        } = req;

        if (!validationResult.isValid) {
            return ResponseHandler.json({
                message: validationResult.message
            }, StatusCodes.BadRequest);
        }

        return ResponseHandler.json({});
    } catch (e) {
        console.error(e);
        return ResponseHandler.json(
            {}, StatusCodes.InternalServerError);
    }
};
```

Listing 2.4. index.ts content

```
export default LambdaHandler
    .bind(this, handler, validations, 'postUsers');
```

Listing 2.5. validations.ts content

```
const queryStringParametersValidations =
(): YupShapeByInterface<QueryStringParameters> => ({});
```

```
const payloadValidations =
(): YupShapeByInterface<Payload> => ({});

export default () => ({
    queryStringParameters: yup.object()
        .shape(queryStringParametersValidations()),
    payload: yup.object()
        .shape(payloadValidations()).noUnknown(),
});
```

Listing 2.6. interfaces.ts content

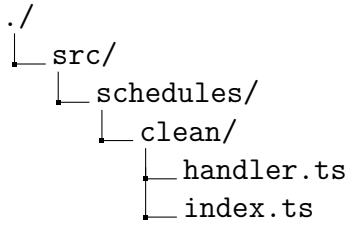
```
import { RequestI } from '@restlessness/core';
export interface QueryStringParameters {}
export interface Payload {}
export interface Request extends
    RequestI<QueryStringParameters, Payload, null> {};
```

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 2.6, and also save the provided information under the config/schedules.json file.

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

Figure 2.6. Structure of a schedule endpoint folder



is used to wrap the handler function, the same way as happens for endpoints, with the purpose of executing the framework lifecycle hooks.

Listing 2.7. handler.ts content

```
export default async (event) => {};
```

Listing 2.8. index.ts content

```
import { ScheduleHandler } from '@restlessness/core';
import handler from './handler';
export default ScheduleHandler.bind(this, handler, 'clean');
```

2.3.3 Models

@todo

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 endpoints.

Listing 2.9. index.test.ts template

```
const postUsers = 'postUsers';

beforeAll(async done => {
    await TestHandler.beforeAll();
    done();
});

describe('postUsers API', () => {
    test('', async (done) => {
        const res = await TestHandler.invokeLambda(
            postUsers);
        // expect(res.statusCode).toBe(StatusCodes.OK);
        done();
    });
});

afterAll(async done => {
    await TestHandler.afterAll();
    done();
});
```

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 (listing 2.10), and a complementary remove command that removes all the services enforcing an opposite ordering.

Listing 2.10. Deploy command

```
$ restlessness deploy
$ restlessness deploy --env production
$ restlessness deploy --env production users
```

It is possible to deploy the application on different environments, 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.

Since the deploy operation involves more than one service, it's important that the information between them are consistent, especially when deploying. This is

why the deploy command, under the hood, takes care of performing this check, with a method from the JsonServices class, named healthCheck. In particular, it checks that the various services belong to the same serverless organization and organization, the same aws deploy region, and that do not exist services with functions associated to the same path. The latter is due to the fact that the services use a 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 information, 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 environments: 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 shown on listing 2.11.

Listing 2.11. Environment variable syntax

```
key1=${otherKey}
key2=sample ${key1}
```

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 functionalities using external packages. To achieve this, has been defined an AddOnPackage class, containing the following lifecycle hooks:

- postInstall: executed after the addon package has been installed. Here it's possible to perform initialization 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 initialization, 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 Data Access Object

To simplify the creation of a Data Access Object, Restlessness provides the abstract class DaoPackage (listing 3.1), which extends the AddOnPackage class previously defined.

Listing 3.1. DaoPackage class definition

```
abstract class DaoPackage extends AddOnPackage {
    abstract modelTemplate(modelName: string): string
}
```

In addition to the previously defined 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.

3.1.1 Dao for mongodb

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 base class containing the main Dao functionalities for CRUD operations, as shown on listing 3.2.

Listing 3.2. MongoBase class definition

```
export default class MongoBase {
    _id: ObjectId
    created: Date
    lastEdit: Date

    static get collectionName()
```

```
    static get dao(): MongoDao
    async getById(_id: ObjectId): Promise<boolean>
    static async getList<T>(
      query: QuerySelector<T> = {},
      limit: number = 10,
      skip: number = 0,
      sortBy: string = null,
      asc: boolean = true
    ): Promise<T[]>
    static async getCounter<T>(
        query: QuerySelector<T> = {}): Promise<number>
    async save()
    async update()
    async patch(fields: any): Promise<boolean>
    async remove<T>(): Promise<boolean>
    static async createIndex(
        keys, options): Promise<boolean>
}
```

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 listing 3.3.

Listing 3.3. modelTemplate function definition

```
const modelTemplate = (name: string): string => `
import {
    MongoBase, ObjectId
} from '@restlessness/dao-mongo';
```

```
export default class ${name} extends MongoBase {
    ['constructor']: typeof ${name}

    static get collectionName() {
        return '${pluralize(name, 2).toLowerCase()}';
    }
};
`;
```

Database Proxy

The MongoBase class uses the MongoDao class internally to perform database operations. The latter class, at the early stage of Restlessness development, offered an abstraction layer over the official mongodb driver for Node.js, effectively using the driver internally. As described on chapter 5, this approach showed its drawbacks in the context of a serverless application, so the next approach has been to exploit the concept of Database Proxy. The main idea is to have a serverless function, the proxy, with the task of performing all database access, on behalf of all other serverless functions. Another advantage of Serverless is indeed the possibility to invoke a function from another one, but this comes at the cost of a doubled Cold start (5.1.1), resulting in a performance degradation for some requests. However the solution provided on 5.1.1 is particularly useful in this case because enabling the warmup plugin on the proxy function, avoids the costs of function initialization and also database connection, making it possible to enable warmup only on a small group of functions, so the overall performance improves or stays the same.

To implement this structure has been developed a serverless plugin, named serverless-mongo-proxy, and usable independently of the Restlessness framework.

The plugin automatically creates the serverless proxy function in the specified service, which in the case of Restlessness is the shared one, so all services can exploit the advantages of using a proxy. Since all informations exchanged between serverless functions must be serialized, the plugin used the bson encoding, to obtain consistent representation for data types such as dates and regular expressions.

The MongoDao class can then invoke the proxy function internally, without having to keep a connection open.

3.1.2 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

Chapter 5

Application

During its development process, the Restlessness framework has been tested on real deployed applications, and this has been fundamental as it helped finding bugs and critical issues at an early stage. The main issues have emerged during the implementation of the backend for the project Spazio alla scuola, a platform thought by the Fondazione Agnelli.

The foundation is a non-profit, independent institute for social science research, born in 1966 in Turin, by the lawyer Agnelli, on the occasion of the centenary of the birth of the founder of Fiat, Senator Giovanni Agnelli. Its purpose is to work in support of scientific research and to disseminate knowledge of the conditions on which Italy's progress depends.

5.1 Spazio alla scuola

The project Spazio alla scuola aims to provide a concrete support to school leaders for lecture resumption on September 2020, given the health situation on the country due to the SARS-CoV-2 pandemic. The platform offer tools to verify capacity of classroom and other school spaces, to plan classrooms flows and staggering, in compliance with the distancing measures. The platform is provided as a free service

and is available at the address www.spazioallascuola.it.

5.1.1 Cold start

The first encountered problem has been Cold start, a new term in the serverless development that denotes the situation in which a serverless function is not active yet, so the platform must perform some resources initialization, with the main one being[1]:

- Code: the project's code is uploaded in a zip archive, so it needs to be downloaded and extracted.
- Extensions: aws allow associate extensions to a lambda function, to integrate it with custom monitoring, security or other tools.
- Runtime: bootstrap operation for the chosen runtime environment, it is also possible to provides a custom runtime if needed.
- Function: code written by the developer, it can perform some resource initialization, such as creating a database connection.

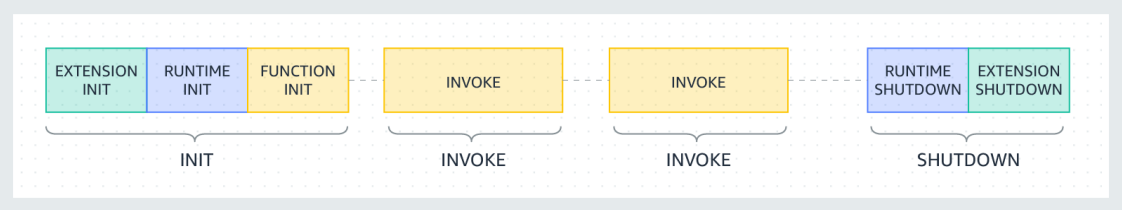


Figure 5.1. Aws Lambda lifecycle

The Cold start refers exactly to this Init phase and represent an overhead to the function execution, however once this phase is completed the function is ready,

[1]Relatively to the Aws platform

and subsequent invocations will not suffer from it. Then after some times without receiving any events, usually in the order of 5 to 20 minutes, the platform performs the Shutdown phase, so any following event causes the process to start again from the Init phase. For the majority of runtimes the duration of the Cold start varies in the order of tenths of a second, as shown on figure 5.2. The provided numbers vary also based on the memory allocated for the function and the size of the provided code package.

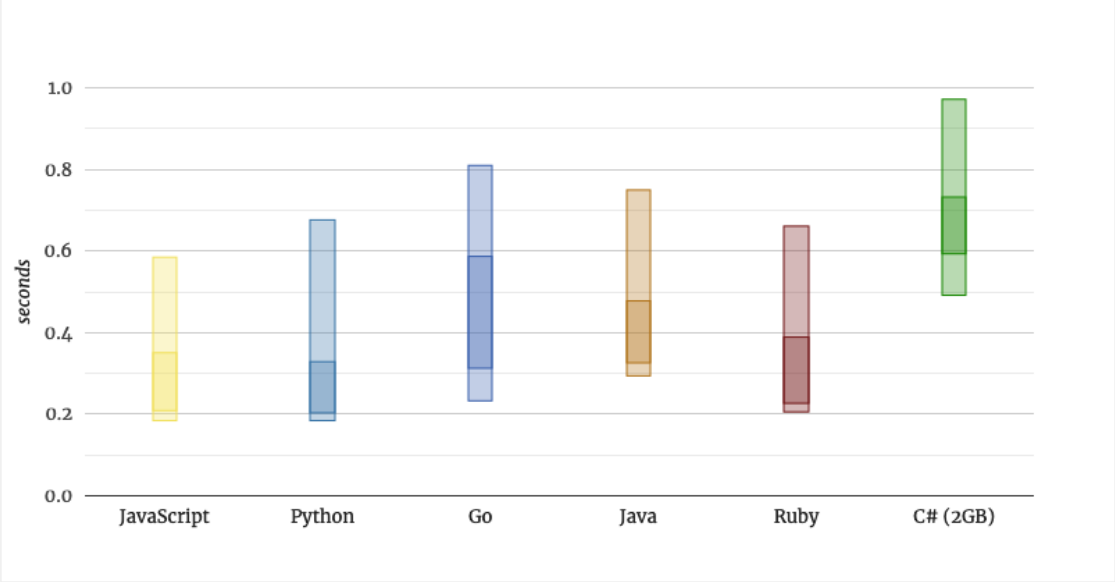


Figure 5.2. Cold start duration for different runtimes

One of the approaches to mitigate the effect of the Cold start proposed by the Serverless community has been the plugin named serverless-plugin-warmup. The plugin creates a scheduled function programmed to invoke the other defined functions, forcing the platform to keep an active container for each function. This way the Cold start effect remains present, but the end user of the api does not experience it.

Has been decided to make this plugin an integral part of the Restlessness framework, granting out of the box support for it. From the Web Interface is possible

to enabled or disable the warmup on the single endpoint, since not all functions may need it. By including the warmup plugin into the framework the effect of Cold start has been mitigated, however it introduced another type of issue.

Figure 5.3. Endpoint resource with warmup enabled

5.1.2 Database proxy

The project Spazio alla scuola rely on the popular non relational database mongodb. As stated previously, each function run in its own runtime, independently from the others, consequently each function requiring database access needs to open a non shared connection. So the number of active connections can become quite high, depending on the number of active functions, furthermore using mongodb, the connection remains active for a certain amount of time even after the function has been shutdown. This leads to a high number of active connections, which is a problem, not only in terms of resources used, since each connection requires memory usage on the database, but also because mongodb has a limit of 500 concurrent connections, and once the threshold is exceeded the application experiences random errors when performing database operations.

Figure 5.4. Mongodb connections reaching the 500 threshold

Although the problem has been amplified by the introduction of the warmup plugin integration, it remains a critical issue for application that rely on the high scalability of the serverless platform. To address this problem on its relational databases, Aws rely on the usage of a proxy between the functions and the database. Exploiting the concept of a proxy, has been decided to approach the problem in the same way, since a solution for for the mongo database does not exist at the moment. Restlessness already provides the package @restlessness/dao-mongo, as described

on section 3.1, defining an abstraction level over the mongodb driver, so it was possible to include a proxy without changing the exposed methods for the users. Has been decided to develop an open source plugin, named serverless-mongo-proxy, to provide the proxy functionality, independently from the Restlessness framework, as shown on 3.1.1. The dao-mongo package then uses the plugin internally, providing an effective solution to the presented problem.

5.1.3 Micro services

During the deployment of an application on the Aws platform a number of resources are created for each function, to provides services such as logging, api gateway for http events, permissions and others functionalities. The Aws platform has a threshold of maximum 200 resources definable for each service (1.3.2), and since for each function there are about 10 resources associated, it follows that each service can define about 20 functions. Since the serverless paradigm proposes a Micro services oriented approach this limitation actually force developers to compose their application as a set of low complexity services. So the next step in the Restlessness framework development has been to switch between the management of a single service, to a multitude of services, under the same Restlessness project (2.1). With this approach has been possible to split the functions of the project Spazio alla scuola into multiple services, obtaining a more fine grained separation between its logic components.

In conclusion the choice of using serverless, combined with the Restlessness framework for the backend api of Spazio alla scuola, brought the desired benefits in terms of ease of development, after the proper framework improvements described previously. At its peak, the api service has managed 500 thousand requests, demonstrating the advantage of the natural scalability of the serverless approach.

Chapter 6

Deployment

6.1 Aws

todo

detailed description of what resources are created when deploying

6.2 Why Aws

Bibliography

[1] What is serverless https://www.cloudflare.com/learning/serverless/what-is-serverless

[2] What is IaaS https://www.cloudflare.com/learning/cloud/what-is-iaas

[3] What is PaaS https://www.cloudflare.com/learning/serverless/glossary/platform-as-a-service-paas

[4] What is SaaS https://www.cloudflare.com/learning/cloud/what-is-saas

[5] Serverless pros and cons https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9

[6] Serverless Framework Aws Guide https://www.serverless.com/framework/docs/providers/aws/guide/intro

[7] Dijkstra, Edsger W (1982). "On the role of scientific thought". Selected writings on Computing: A Personal Perspective. New York, NY, USA: Springer-Verlag. pp. 60–66. ISBN 0-387-90652-5. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

[8] Data Access Object Pattern https://www.oracle.com/java/technologies/dataaccessobject.html

[9] Spazio alla scuola https://www.fondazioneagnelli.it/2020/07/17/spazio-alla-scuola

[10] Aws Lambda Environment https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html