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**

In the scene of services for the creation of web applications is focusing more and more towards a micro services oriented approach, moving away from structures called monolithic. The maximum representation of this is with the serverless paradigm, which since 2014 has seen an ever greater increase in its use and in its investments by the major cloud providers. Such a paradigm has found an implementation in the cloud model Functions as a Service, which uses as its main resources, plain simple functions. Serverless Framework has emerged as one the major framework that allow the usage of the homonym paradigm in a simple way, and introducing a level of abstraction regarding the underlying structure of the chosen cloud provider. Despite the functionalities introduced by Serverless, the developer has to take charge of various operations concerning indirectly the business logic of the application, with the main one being: to structure the code base, to define the various resources through the compilation of a configuration file, to define a unit testing structure, fundamental once the application complexity increases. Furthermore, based on the chosen cloud provider, the developer must find solutions to problems such as Cold start, and limitations in the creation of resources.

The Restlessness framework was born with the goal of improving the user experience of Serverless, providing a standard project and testing structure, a Command Line Interface and a local Web Interface through which is possible to completely manage the project, and with the further goal of minimizing all operations that do not concern directly the application's business logic. The framework is provided as an Open Source package, and with the possibility of extending its functionalities, through the use of addons, some of which are already present, to address common patterns, such as database access or authentication. During the framework development has been possible to test it on real applications, thus allowing to find and

correct critical issues, whose main ones were: Cold start handling, use of the non relational database mongodb, and limitations on the applications structure proposed at the beginning.

# Contents

# Chapter 1

# Cloud services

In the early days of the web, anyone who wanted to build a web application had to buy and maintain the physical hardware required to run a server, which was a cumbersome process to undertake, especially for small businesses [1]. Then came a new paradigm for the provisioning of computing infrastructure, named Cloud Computing, and defined as:

"Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs." [2]

Cloud Computing is possible because of a technology called virtualization, which allows the creation of a simulated computer, named virtual machine, that behaves as if it were a physical computer with its own hardware. When properly implemented, this approach allows having a more efficient use of the physical hardware, as each computer is able to run many virtual machines at once. Despites the many benefits, using virtual machines still requires manual server administration, as each one simulate a full system, including the operating system and the underlying kernel.

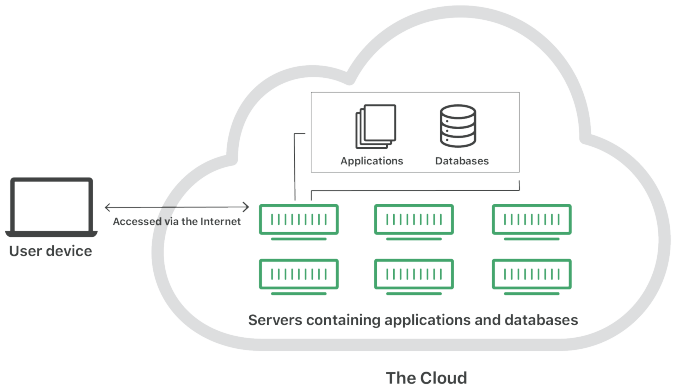



Figure 1.1. Representation of the cloud

The next technological step has been containerization, which gave the possibility of packing an application and all its dependencies, such as system libraries and system settings into a single entity called Container. With this approach a single physical machine, including the kernel, is shared by a multitude of containers. The main advantages that containerization offers, with respect to virtual machines are [3]:

- Portability: once the application is packed into a container it can be run on any host supporting that technology.
- Control and flexibility.
- Faster deploy.
- Less server administration.

With this premises about the cloud and its infrastructure is possible to outline the main models that have emerged in the context of cloud computing.

## 1.1 Cloud computing models

Between the various types of cloud computing architectures have emerged three main models, which are: Infrastructure as a Service, Platform as a Service and Software as a Service. Each model is characterized by an increasing level of abstraction regarding the underlying infrastructure.

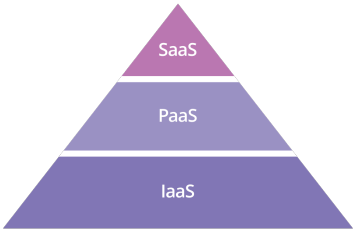



Figure 1.2. IaaS, PaaS, SaaS Pyramid

### 1.1.1 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 physical 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 [4].

### 1.1.2 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 [5]. An example of this kind of services is Google with its product App Engine.

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.

### 1.1.3 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 [6].

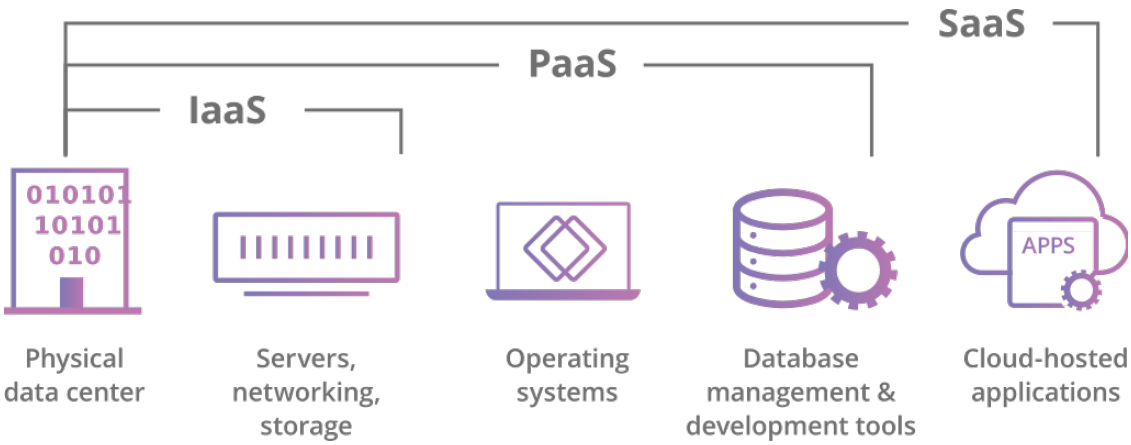



Figure 1.3. IaaS, PaaS, SaaS diagram

## 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 1.4.

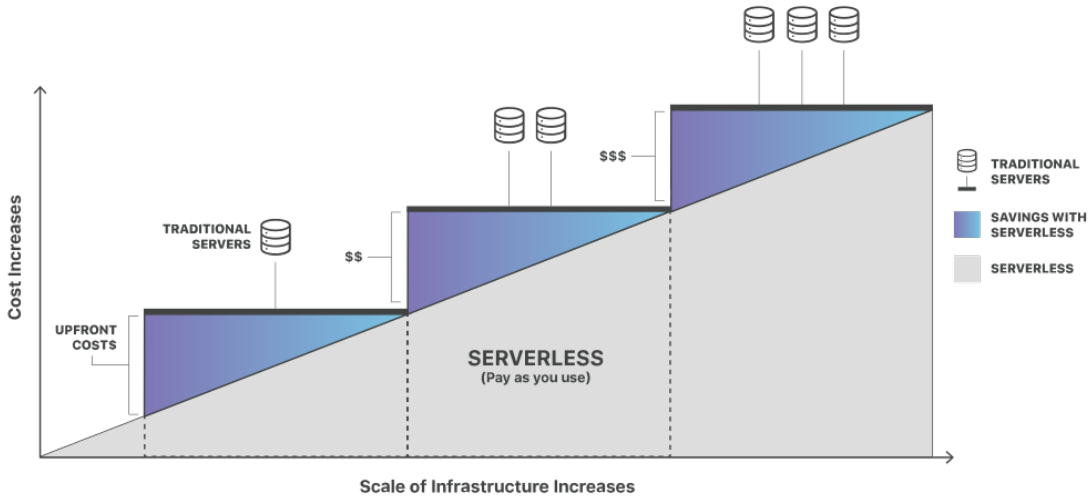



Figure 1.4. Cost Benefits of Serverless

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 (1.5), since the code is organized as a set of independent

functions from the beginning.

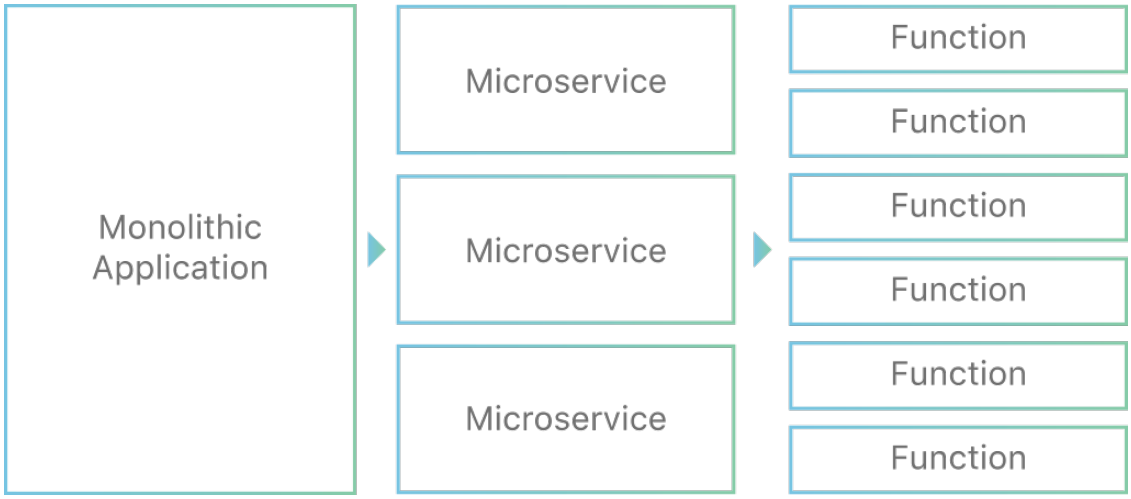



Figure 1.5. Monolithic to Micro services application

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. 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.

Although the serverless framework, given the number of cloud providers supported, aim to be platform agnostic, the following examples will be based on the Aws provider and on the Node.js 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 [7]. Each function is executed only when triggered by and Event, which can be of different type, 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 an application can then be composed by multiple services. This structure reflects the modular approach of the micro services architecture described previously. Finally various applications are grouped under an organization (1.6)

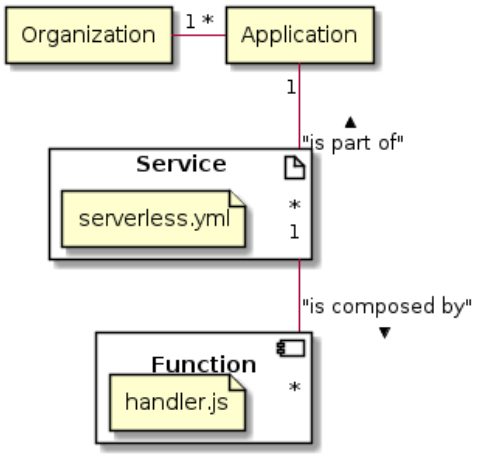



Figure 1.6. Serverless framework resources scheme

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.

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

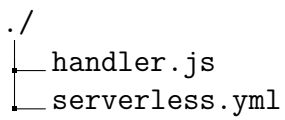
Listing 1.1. Simple serverless.yml file

```
async function usersCreate(event, context) {
    const user = {
        name: 'sample_name',
        surname: 'sample_surname'
    }
    await mockDb.createUser(user)
    return {
        statusCode: 200,
        body: JSON.stringify({user})
```

```
    }
}
```

Listing 1.2. Simple handler function

Figure 1.7. Simple Serverless project structure




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

```
$ serverless deploy
Serverless: Stack update finished...
Service Information
service: users
stage: dev
region: us-east-1
stack: users-dev
resources: 12
api keys:
  None
endpoints:
  POST - https://.../dev/users/create
functions:
  usersCreate: users-dev-usersCreate
layers:
  None
```

Listing 1.3. Deploy command

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, 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

```
$ serverless invoke -f usersCreate
{
    "statusCode": 200,
    "body": "{\"user\":{\"name\":\"sample_name\", ...}}"
}
```

Listing 1.4. Invoke command

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 Conclusions

Each cloud model presented has its own strength and drawbacks, depending on the needs of the wanted goal. Favouring as selection criteria, solutions that present major advantages in terms of scalability, cost efficiency and speed of development, has been decided to favour the Serverless option. The main cloud providers offering this kind of service, as previously stated, are: Aws, with its Lambda service, Microsoft, with Azure Functions, and Google, with Cloud Functions. Each provider offer different configurations, with different pricing, based on memory, CPU, and execution time as parameters, as shown on 1.1. In the literature there are several documents comparing the various services side by side exhaustively [8]. For the project subject of this document has been chosen Aws as the main provider, as the most mature platform meeting the project's needs. In particular it providers the following advantages with respect to the competitors [8]:

- Cold start (1.2)
- Overall maturity
- Performance consistency
- Scalability

| | **AWS** | **Azure** | **Google** |
|---|---|---|---|
| Memory (MB) | 64 * k (k = 2, 3, ..., 24) | 1536 | 128 * k (k = 1, 2, 4, 8, 16) |
| CPU | Proportional to Memory | Unknown | Proportional to Memory |
| Language | Python Nodejs Java, and others | Nodejs Python, and others | Nodejs |
| Runtime OS | Amazon Linux | Windows 10 | Debian 8 |
| Local disk (MB) | 512 | 500 | > 512 |
| Run native code | Yes | Yes | Yes |
| Timeout (second) | 300 | 600 | 540 |
| Billing factor | Execution time, Allocated memory | Execution time, Consumed memory | Execution time, Allocated memory, Allocated CPU |

Table 1.1. Cloud providers configuration [8]

| Provider-Memory | Median | Min | Max | STD |
|---|---|---|---|---|
| AWS-128 | 265.21 | 189.87 | 7048.42 | 354.43 |
| AWS-1536 | 250.07 | 187.97 | 5368.31 | 273.63 |
| Google-128 | 493.04 | 268.5 | 2803.8 | 345.8 |
| Google-2048 | 110.77 | 52.66 | 1407.76 | 124.3 |
| Azure | 3640.02 | 431.58 | 45772.06 | 5110.12 |

Table 1.2. Cloud providers Cold start (in ms) [8]

# Chapter 2

# Tools

An important process in the software development is the choice of the right tools, in order to achieve simplicity and efficiency for both development process and the project itself. In this chapter will be described the main tools used during the development of Restlessness and its deployment to make its use available to everyone.

## 2.1 JavaScript

JavaScript is a lightweight interpreted programming language. Interpreted means that the code is read top to bottom, and the result of the running code is immediately returned. Interpreted programming languages are opposed to compiled one, where the code is transformed into a binary format that can be directly executed [9]. Although JavaScript was born as a language limited to client side programming, exploiting an engine directly incorporated into the Web browser, with the introduction of Node.js has become possible to use this language also for back-end programming, and in general in contexts outside of the browser. Node.js is a JavaScript runtime based on the V8 engine, on which the popular Chrome browser is based [10]. A key characteristic and one of the main strength of JavaScript

with respect to other programming languages is its asynchronous nature, that allows having non-blocking I/O. As a consequence of this the code runs on a single thread, based on a LIFO queue (Last In, First Out) continuously checked by the so called Event Loop. As shown on 2.1, operation regarding File System, Network or Database access are executed separately and only once completed are inserted again into the queue, to handle their result, meanwhile other queued code is executed by the only present thread.

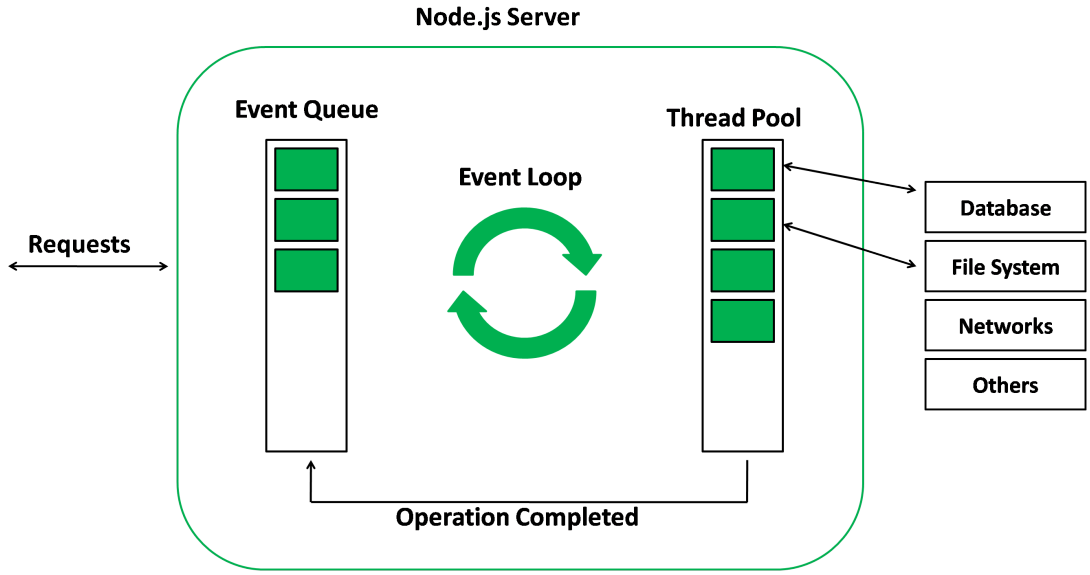



Figure 2.1. The Event Loop [13]

Being single threaded is a useful limitation as it's not possible to incur into concurrency issues. This peculiarities make JavaScript well suited for the so called real-time applications (RTAs), that is applications that have to process a high volume of short messages requiring low latency, and so they require a highly scalable solution. Conversely due to its single threaded nature, JavaScript is not recommended for CPU-heavy jobs, as the Event Loop would be stuck on a single operation [11][12].

Another advantage of JavaScript, especially after the release of Node.js, is the possibility to use the language for both frontend and backend in the context of Web

development, creating a seamless experience for developers.

JavaScript is a dynamically typed language, which means that it's not necessary to explicitly mention the type of data a variable holds, as that type can change dynamically as change the content of the variable (2.1).

```
let a = "Hello World!"
a = 42
```

Listing 2.1. Dynamically typed variables

This feature of the language gives a lot of flexibility to developers, however as the project complexity grows it can quickly become a downside. For this reason, in 2012 Microsoft released an open source language called Typescript, a superset of JavaScript that enable static type checking. Being a superset, any JavaScript code is also valid Typescript code, enabling a gradual integration for already existing code bases. The Typescript compiler is specifically a transpiler, or a compiler that take source code as input and produces other source code as output, in this case JavaScript code. The compiler will point the errors it encounters, but it does not prevent the code to be run, hence it behaves like a spellchecker for the code. Typescript can also infer variables type from their usage, reducing the effort needed to enable static type checking from the developer [14][15].

```
interface Student {
    name: string
    graduationYear: number
}

const aStudent: Student = {
    name: 'Arthur Dent',
    graduationYear: 2020
}
```

```
aStudent.graduationYear = '2020'     // Invalid
aStudent.graduationYear = 2021       // Valid
```

Listing 2.2. Static type checking on Typescript

Keeping in mind the described strengths of the JavaScript environment, has been decided to use it as the main language for the development of the Restlessness framework.

## 2.2 Npm

The strengths of the JavaScript ecosystem are further increased by the presence of Npm, shorthand for Node Package Manager, which is the official package Manager for Node.js. Npm rely on the commonjs modules specification [16], which defines a convention for the JavaScript module ecosystem. The main components of Npm are:

- Npm registry: modules can be published to it or installed from it. The official and main registry is available at the address `https://npmjs.org`
- *npm* CLI: the command line tool from which is possible to interact with the registry, with operations like publishing or installing packages.
- *package.json*: a configuration file, in the Json format [17], that must be present for both modules that are published into the registry and modules that use other modules from the registry as dependencies. It contains projects information, such as name and version, and a list of other modules, on which the project depends on.
- *node_modules*: an automatically created folder that contains all the projects dependencies. At runtime Node.js looks for modules in this folder.

```
function add(n1, n2) {
    return n1 + n2
}

module.exports = add
```

Listing 2.3. CommonJs module definition

```
const add = require('./add.js')
```

Listing 2.4. CommonJs module usage

```
{
    "name": "simple_module",
    "version": "1.0.0",
    "description": "Simple module example",
    "main": "index.js",
    "author": "Arthur Dent",
    "license": "ISC"
}
```

Listing 2.5. A simple *package.json*

The Npm ecosystem has been used extensively during the development of Restlessness, for its dependencies and for making it available on the registry. Furthermore, the developed framework uses a feature of Npm called Scoped Packages [18], which allows to group related packages together under a common scope, acting as a namespace. Restlessness packages are available under the *@restlessness/* scope.

## 2.3 Github

### 2.3.1 Git

Git is an Open Source Distributed Version Control System, in particular:

- Control System: Git is a content tracker, it can be used to store content, which generally is code.
- Version: the tracked content is subject to continuous change, often this changes are added in parallel. Git helps handling this by maintaining a history of all changes.
- Distributed: Git is based on remote and local repositories, the first one stored in a server, while the latter is stored in the developer computer, and both contains the full history information.

Git is useful to track code changes in all cases, but it's absolutely necessary to avoid conflicts when multiple developers work in parallel on a single codebase. The main concepts introduced by Git are:

- Commit: the main unit representing content modification.
- Branches: allows working simultaneously at the codebase, making different modifications.
- Push/Pull: operations that allow synchronization between the remote repository and the local one.
- Merge: operation that integrate the modification made on a branch into another branch.
- Tag: a string identifier assigned to a specific commit, useful to reference a particular version of the project (e.g. a simple tag is *v1.0.2*).

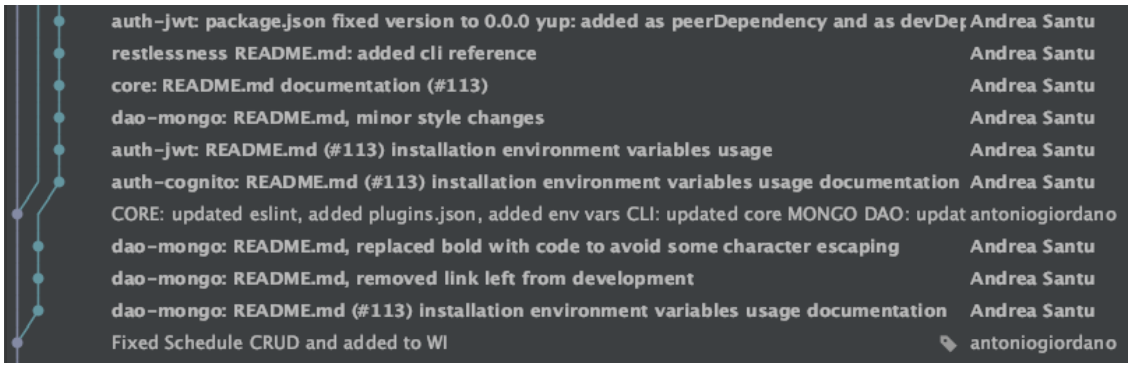



Figure 2.2. Section of Restlessness history

With these concepts it is possible to work on each feature independently from others, integrating it only when it reaches an appropriate stability level. The strategy adopted with the developed framework has been to create branches with the *feature/* prefix for new functionalities or improvement of existing ones, and the *fix/* prefix for correction of bugs, followed by the name of the specific feature of fix.

### 2.3.2 Github features

Github is a web based platform providing all functionalities offered by the Git system plus additional DevOps features, with the main ones used during the development of Restlessness being: Issues, Pull Requests and Projects.

#### Issues

Issues are Github feature that helps to keep track of tasks, bugs, enhancements or any kind of modification to the project. They are characterized by a title, that gives an immediate feedback about what is the reason of the Issue, and an optional description, with more specific and technical information, as shown on figure 2.3. Each Issue can be assigned to one or more collaborators, responsible for having it solved. This tracking system is focused on collaboration, as it is possible to comment and discuss about the Issue with other collaborators, also referencing

other resources, which can be other Issues or code sections. As the project grows so does the number of Issues, and so become important to keep them organized. This is made possible by using Labels and Milestones. Both allow to group Issues according to a common characteristic, but with a different granularity [19]. The first one allows a more specific grouping, with the main ones defined for Restlessness being:

- *enhancement*: A new feature, or a request for a new feature.
- *bug*: A problem in the project functionalities.
- *documentation*: Improvements or additions to documentation.
- *tests*: Testing related Issues
- *good first issue*: Being the framework Open Source, also external people can contribute to it, this Label marks simple and easy Issues that can be managed also by newcomers.
- Packages specific Issues: Restlessness adopt a monorepo strategy [20], having all provided packages under the same repository, so it has been defined a Label for each package, such as: *CORE*, *CLI*, *AUTH-cognito* and *DAO-mongo*.

The latter instead group together Issues linked together from a temporal point of view, typically a version release or a planned Sprint if following the agile methodology [21]. With the Restlessness framework has been opted for the first option.

### Pull Requests

An important process when multiple developers collaborate on a single project are code reviews, as having project's modification verified by more than one person reduces the risk for finding bugs, typos and critical problems later. Pull Requests are a feature of Github that enable this process, with it a collaborator proposes its

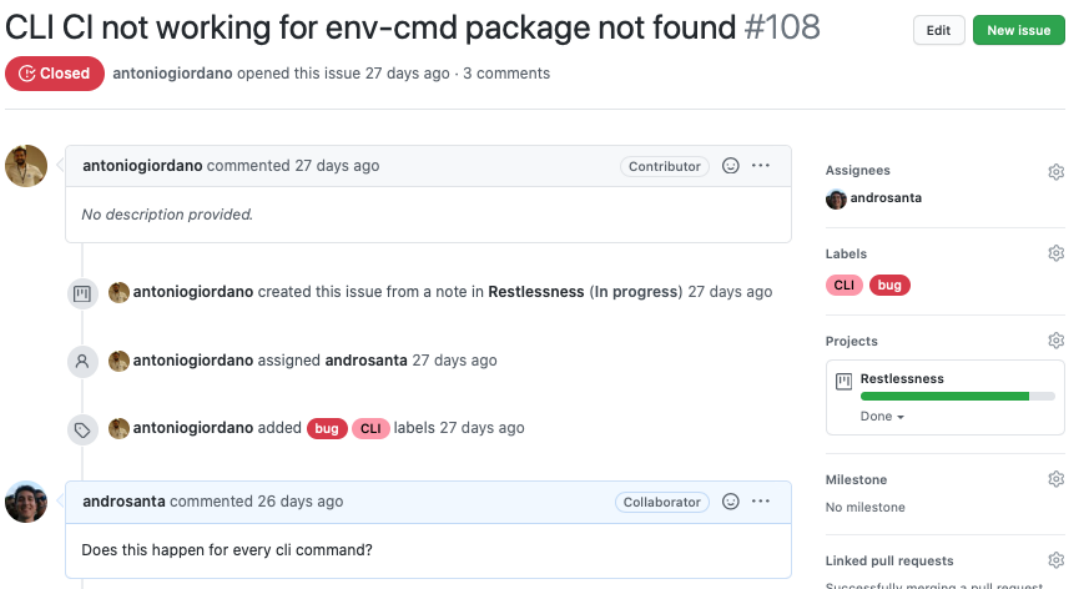



Figure 2.3. An Issue on the Restlessness project

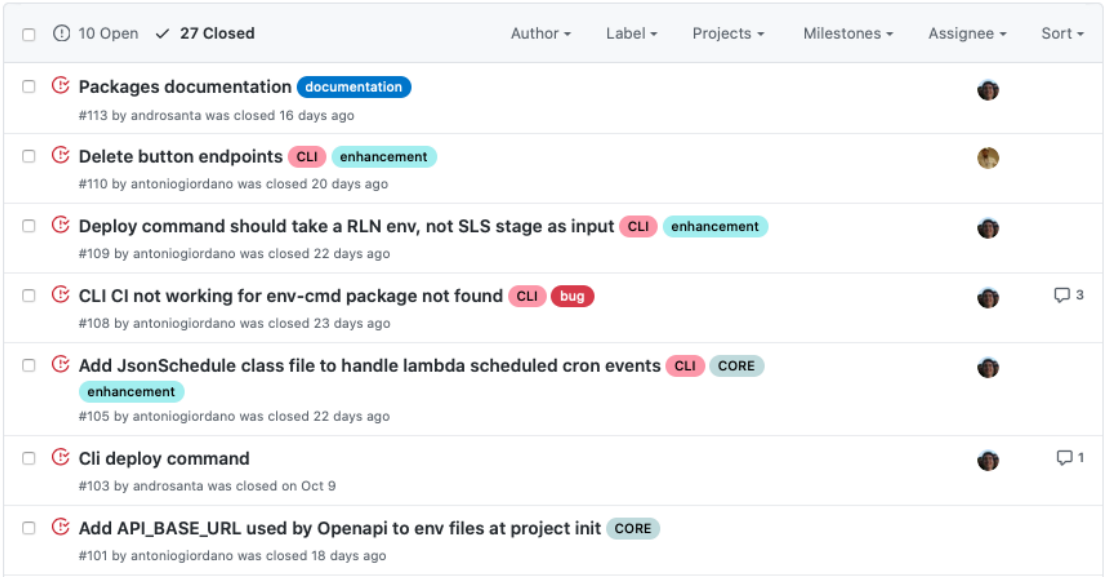



Figure 2.4. List of closed Restlessness Issues

changes while another one accept or reject the request. It is possible to discuss on the specific request, referencing other resources, commenting on code or requesting

modification on the proposed changes, as it happens for Issues. When a Pull Request is created the author chose a target branch on which to integrate its proposed changes, and once the request is accepted those changes are merged into the target branch, and the Pull Request is considered closed, as shown on figure 2.5.

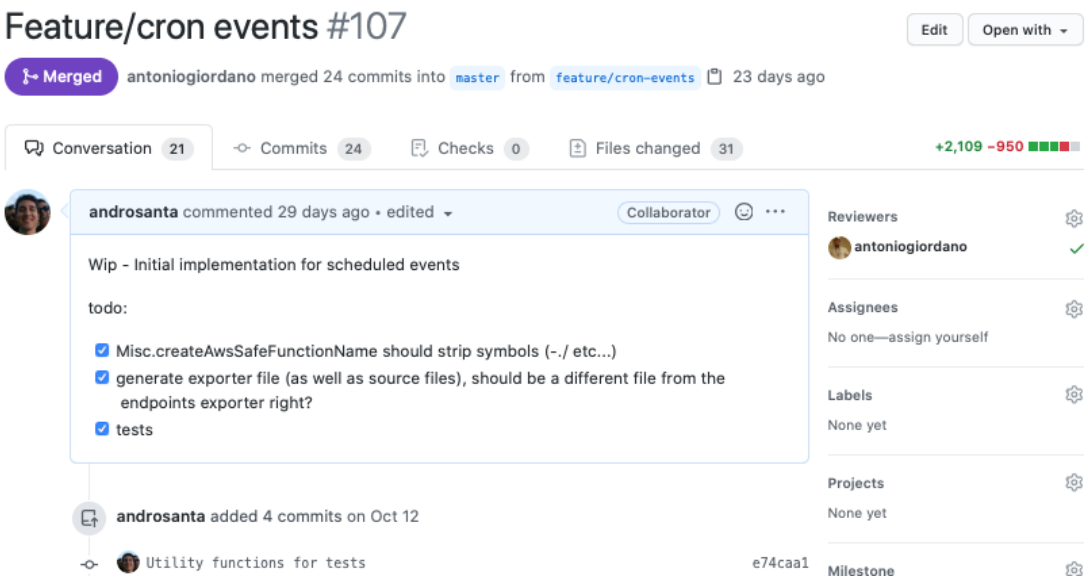



Figure 2.5. An approved Pull Request on the Restlessness project

### Projects

Projects is a recently added Github feature with the purpose of further improve organizing and distributing tasks and work. From the Projects page is possible to define custom columns in which assign different tasks, which can be Issues, Pull Requests or simple Notes. As shown in figure 2.6, for Restlessness has been defined tree columns: *To do*, *In Progress* and *Done*. This way it is immediately visible which tasks need to be done, are under development or are already completed.

Being the developed framework Open Source, it is available for consultation, modification and improvement on Github, as well as this document, on the following

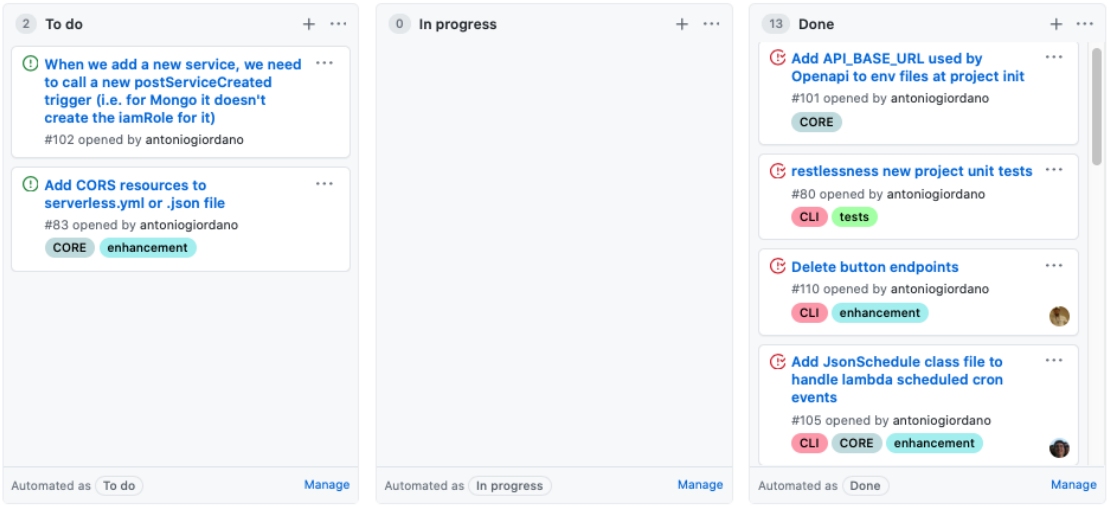



Figure 2.6. Github Projects board on Restlessness

addresses:

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

## 2.4 CircleCi

### 2.4.1 CI/CD

Continuous Integration is a practice that encourages developers to integrate their code changes early and often, into the main and stable version of the project, which for a git based project is the *master* branch. Each code integration triggers an automated build and test, that if failed can be repaired quickly. The main advantage of using this approach is the early bug detection, which as consequence will result in an overall reduced bug count and reduced maintenance. Moreover once set, the CI process does not add any overhead to the development as it is completely automated. The CI approach is oftentimes related to another approach, which

is the Continuous Delivery, defined as " Continuous Delivery (CD) is a software engineering approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time." [22]

### 2.4.2 The platform

CircleCi is an online platform that provides services for implementing Continuous Integration and Continuous Delivery (CI/CD) on software projects. It can be configured to access the source code repository on Github, and after that each commit can trigger an automated build, test and deploy task. Those automated tasks are performed inside a clean container or Virtual Machine, ensuring a reproducible environment.

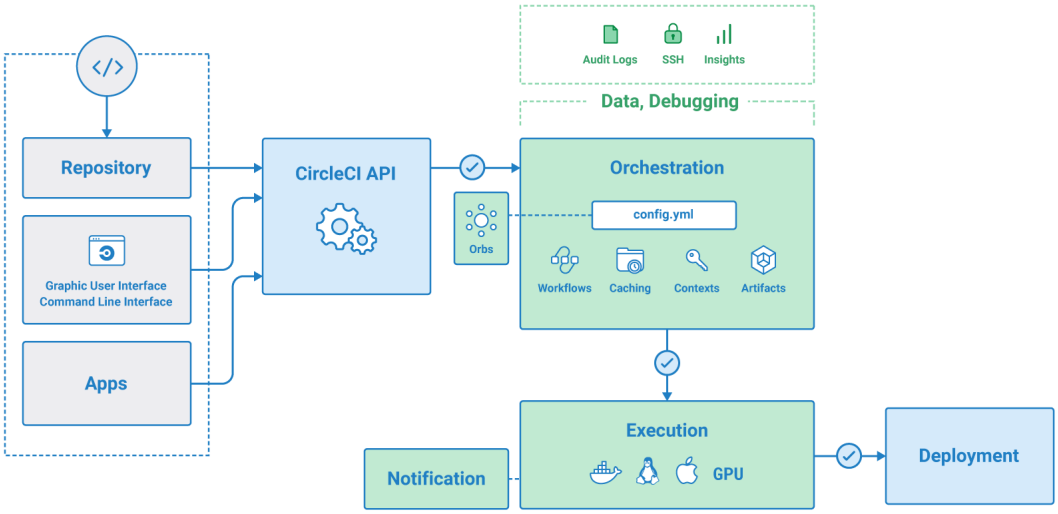



Figure 2.7. CircleCi flow [23]

The main concepts introduced by the platform are:

- Configuration: All processes are orchestrated through a single file called *config.yml*, in the Yaml format, and placed under a folder called *.circleci* at the root of the project.

- Orbs: Reusable snippets of code that help automate repeated processes
- Jobs: Building blocks of the configuration file, they are a collection of steps, which run commands or scripts as specified. Each Job is run in a unique executor.
- Executor: The container or Virtual Machine where running each Job. It is possible to chose between Docker containers, Virtual Machines running Linux, Windows or MacOS.
- Steps: Actions that need to be taken to complete a Job. It can be any kind of executable command.
- Workflows: They define a list of Jobs and their run order, and concurrency.

For the Restlessness development has been chosen the popular containerization solution Docker, in particular a Node.js based container, as shown on listing 2.6:

```
executors:
  node12:
    docker:
      - image: circleci/node:12.9.1
```

Listing 2.6. Reusable executor definition

As previously said the framework adopt a monorepo structure, so has been necessary to define multiple Workflows, one for each package. Each Workflow defines two parallel Jobs, for testing and publishing on the Npm registry. Figure 2.8 show the described structure for two Restlessness packages, and it is possible to notice that each Job run in its own container, in parallel and independently from the others

To perform the Steps shown on 2.8 has been defined reusable commands, with the main one being:

- *install_packages*: Install dependencies as specified by the *package.json*.

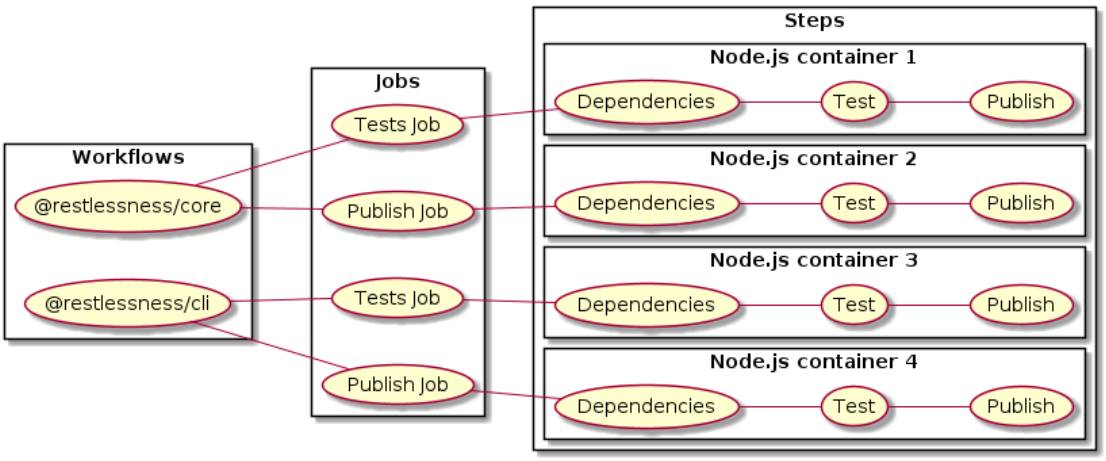



Figure 2.8. CircleCi workflows for Restlessness

- *deps_and_tests*: Install dependencies and run tests as specified by the *package.json*.

- *npm_publish*: Publish the package on the Npm registry.

According to the Continuous Delivery approach the publish operation is triggered manually by performing a git tag on a specific repository commit, following the format: package name, followed by */v* and the semantic version of the package (e.g. *@restlessness/core/v1.0.2*). A custom script takes care of extractive the version information and set it on the correct *package.json*, where is read from the npm publish command.

Although CircleCi offers its own website from which is possible to check Workflow execution, errors and details of every operation, it offers also a Github plugin, that is able to show Workflows result directly on commits or Pull Requests, as shown on 2.9. The integration between the two services has simplified the development workflow of Restlessness, and adds to the already described advantages of adopting a CI/CD approach.

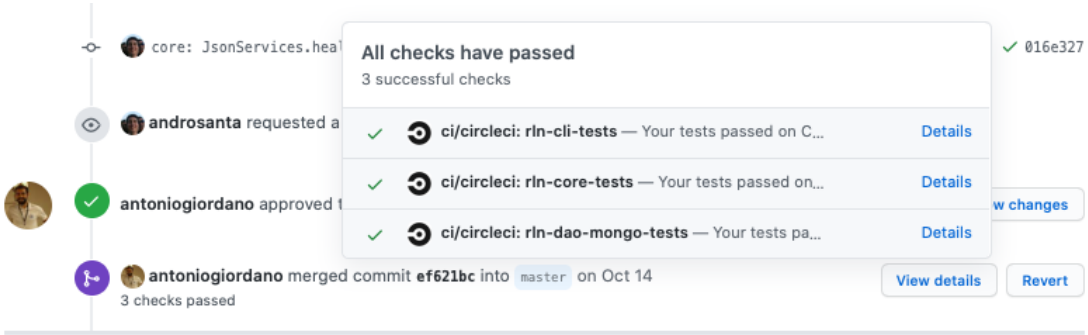



Figure 2.9. CircleCi Workflows seen from Github

## 2.5 AWS

Amazon Web Services is a cloud platform offered by *Amazon.com, Inc* which, between its various services, also provides serverless computing options. Although one of the purpose of using the Serverless Framework is to abstract the underlying infrastructure details of the platform, those details are needed to develop a framework such as Restlessness, that has to interact with the platform at a lower level to provide its functionalities.

Here is a list of the main services used by Serverless and Restlessness on behalf of the user and also used during Restlessness development:

- Lambda: The compute service providing the serverless functionalities. A Lambda function contains the code written by the developer.
- API Gateway: A service that creates a connection point between external requests and other internal services, such as a Lambda function.
- S3: Acronym for Simple Storage Service, provides object storage. Resources are organized in container called Buckets.
- CloudFormation: A service that allow to model infrastructure as code. Each CloudFormation configuration corresponds to a resource called CloudFormation Stack, containing the description of other resources, such as AWS Lambda

functions, API Gateway, and how such resources may interact.

- CloudWatch: A services for monitoring and observability.
- IAM:Acronym for Identity and Access Management, enables the management of AWS resources access.

### Resource creation during deploy

During the deployment of a Serverless service the user code and its dependencies are packaged into a zip artifact, then begin the remote resource creation of a CloudFormation Stack and an S3 Bucket. One that resources initialization has been completed, the CloudFormation configuration and the zip artifact are uploaded and saved into the S3 Bucket and that operation is followed by the creation of all resources defined on the CloudFormation Stack. Those operations are shown on figure 2.10

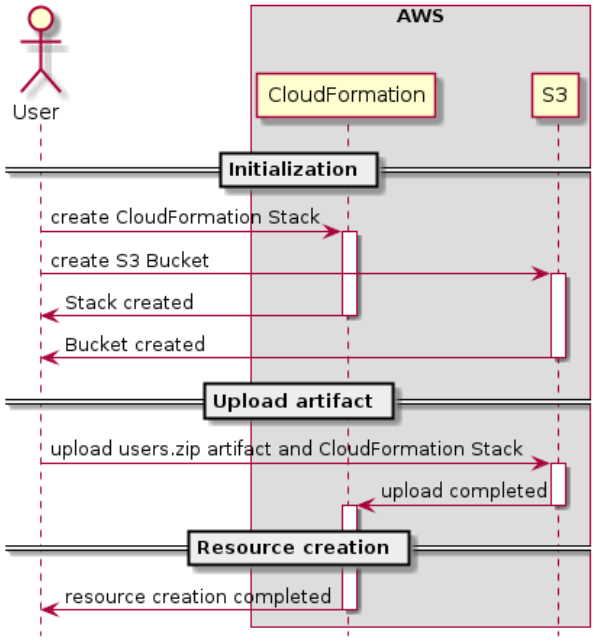



Figure 2.10. Resources creation on Serverless deploy for a *User* service

### Lambda function invocation through an API Gateway

Figure 2.11 shows the simplest possible case of execution flow of an http request, handled by an API Gateway, and forwarded to the Lambda function mapped to the user specified endpoint path.

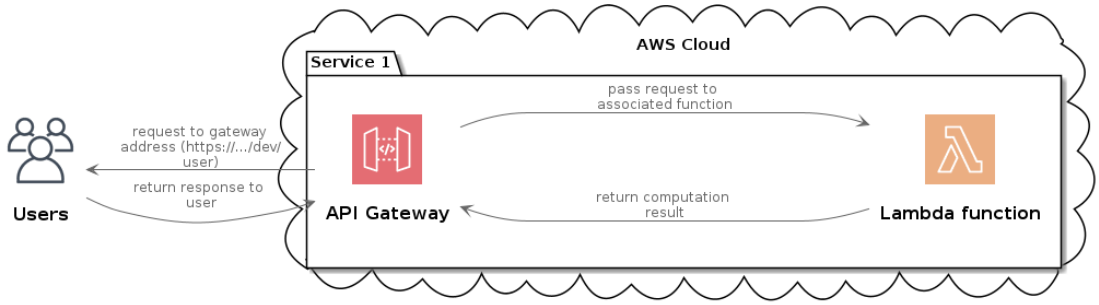



Figure 2.11. Simple Lambda function execution through an API Gateway

A more complex case is given when implementing user Authentication and hence restricting Lambda execution. The Authentication process is made simple by delegating the operation of granting or denying Authentication to a Lambda function, called Lambda Authorizer [24], as shown on figure 2.12. There can be two types of Lambda Authorizers:

- TOKEN: the Lambda receives the caller's identity in a bearer token.
- REQUEST: the Lambda receives the caller's identity in a combination of headers and query string parameters.

The API Gateway forward the request to the specified Lambda Authorizer, that checks the caller's identity and generate an Authentication Policy, which is an object that states which resources the user is authorized to access. The Policy is then cached to improve performance on subsequent requests, and if it the access request is granted, the flow proceed as in the previously described case.

Serverless abstract this structure by allowing to specify a function as Authorizer

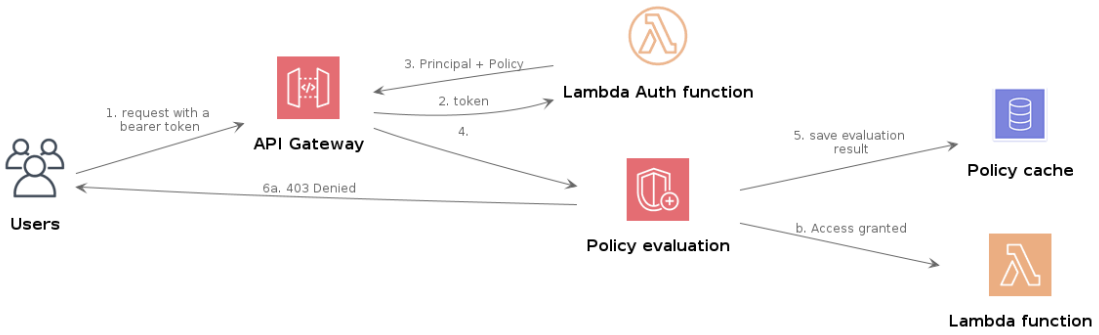



Figure 2.12. Lambda Authorizer function, based on TOKEN identity

of another function, as shown on listing 2.7, where the *getUsers* function is executed only if the function *auth* grants access.

```
functions:
  auth:
    handler: auth.customAuth  # auth.js
  getUsers:
    handler: users.getUsers   # users.js
    events:
      - http:
        path: hello
        method: get
        authorizer: auth
```

Listing 2.7. Authorizer definition on Serverless

## 2.6 React

An important part of the Restlessness framework is its Graphical User Interface, which is the main interaction point for the user. The Frontend development, specifically toward Web Interfaces, can count on the presence of several libraries and

frameworks based on the JavaScript language. For the development of Restlessness has been chosed the popular library React, due to its simplicity, and effectiveness.

React is an Open Source JavaScript library that implements the concept of virtual DOM (Document Object Model) [26]. The browser creates a DOM object at page loading, and then each Html object inside the DOM can be manipulated using JavaScript functions, giving the user an immediate feedback. React instead adopt a different approach by creating a virtual DOM alongside the real one. The virtual DOM is not directly synched with the real one, so it can be modified much faster, not having to reflect those modification on the screen. After those virtual DOM updates are created using the React api, the new istance of the virtual DOM is compared to the previous one, allowing to reflect the update on the real DOM only for the elements that actually change. The library allow to create a structure based on reusable component, obtaining a scalable structure, and is particularly suited for SPA (Single Page Applications) [25]. The library also introduced a new syntax, named JSX (JavaScript XML), and listing 2.8 show the definition of a React component.

```
import React from 'react';

const Card = ({name}) => {
  return (
      <div>{name}</div>
  );
};
```

Listing 2.8. React component definition

# Bibliography


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

[2] A Break in the Clouds: Towards a Cloud Definition ....

[3] What is the cloud https://www.cloudflare.com/learning/cloud/what-is-the-cloud/

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

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

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

[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] Liang Wang, UW-Madison; Mengyuan Li and Yinqian Zhang, The Ohio State University, Thomas Ristenpart, Cornell Tech; Michael Swift, UW-Madison "Peeking Behind the Curtains of Serverless Platforms" ISBN 978-1-939133-02-1 https://www.usenix.org/conference/atc18/presentation/wang-liang

[9] What is JavaScript https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript

[10] Node.js https://nodejs.org/en

[11] Node.js Event Loop https://nodejs.dev/learn/the-nodejs-event-loop

[12] Node.js on Backend https://www.netguru.com/blog/node-js-backend

[13] Node.js Event Loop https://www.geeksforgeeks.org/node-js-event-loop/

[14] TypeScript https://www.typescriptlang.org/

[15] TypeScript: What is it and when is it useful? https://medium.com/front-end-weekly/typescript-what-is-it-when-is-it-useful-c4c41b5c4ae7

[16] Eric Elliott (26 June 2014). Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries. "O'Reilly Media, Inc.". pp. 87–. ISBN 978-1-4919-5027-2. https://books.google.com/books?id=jUfnAwAAQBAJ&pg=PA87

[17] JSON data interchange syntax ISO https://www.iso.org/standard/71616.html

[18] Npm Scoped packages https://docs.npmjs.com/creating-and-publishing-an-organization-scoped-package

[19] Mastering Issues https://guides.github.com/features/issues

[20] From Monolith to Monorepo https://medium.com/@brockreece/from-monolith-to-monorepo-19d78ffe9175

[21] Agile Manifesto https://agilemanifesto.org/iso/it/principles.html

[22] L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too," in IEEE Software, vol. 32, no. 2, pp. 50-54, Mar.-Apr. 2015, doi: 10.1109/MS.2015.27. https://ieeexplore.ieee.org/document/7006384

[23] CircleCi Documentation https://circleci.com/docs/2.0/about-circleci

[24] API Gateway Lambda Authorizers https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-use-lambda-authorizer.html

[25] React - A JavaScript library for building user interfaces https://reactjs.org/

[26] DOM Living Standard https://dom.spec.whatwg.org/

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

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

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

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

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