POLITECNICO DI TORINO

Corso di Laurea Magistrale
in INGEGNERIA INFORMATICA (COMPUTER ENGINEERING)

Tesi di Laurea Magistrale

Designing a Microservice-oriented application running on a Serverless architecture

Relatore
prof. Giorgio Bruno

firma del relatore
.....

Candidato
Enrico Balsamo

firma del candidato
……

A.A. 2017/2018

A one-man army, often a trope of action films,
a heavily armed and well-trained combatant
able to face numerous enemies alone

Summary

The goal of this work is to explore the distributed application architecture indicated by the caption Microservices, and in particular the deployment type named serverless. To make it possible the environment chosen to design, develop, deploy and test the components of such an architecture is the one provided by Amazon and its Web Services (AWS), through the creation of a prototypical application.

The document is structured and developed following a top-down, breadth-first approach. This means that an argument may be talked about in all the chapters, according to the topic the chapter deals with, at an increasing level of specificity, up to the final implementation in the prototype, along with the considerations that need to be performed during all the phases of a software's realization.

Behind the scenes, in fact, there has been a remarkable work in all the phases a software product lives up to the release, described in the last chapter. The final result is a product with its limits due to the limited dimensions of the development team, but almost production ready, that has various advantages, most of all its limited development cost and flexibility.

The conclusions are straightforwardly positive, in fact there is little or no space for not choosing to follow this approach to create a modern infrastructure for the application it is going to be designed: if correctly approached, this architecture is flexible, cost-effective, scalable with the team size and it matches the organization's structure.

Index

Introduction

Information technology's progress through years have always been server-centred.
First, the mainframes were created: big and expensive machines were all the business logic happened that provisioned data ready to be consumed. In this environment, when it comes to distributed web applications for example, an architecture prominently MVC raised, with the creation and diffusion of PHP and JSP among others.
Virtualization came along, and with it the possibility to create virtual systems providing data just like a server but entirely built as a software: software started mimicking servers.
Then, as familiarity with such systems grew, it was time for some big companies having big data centers to provide virtual systems on demands: this paradigm is called Infrastructure as a Service, and it deals with the distribution, indeed as a service, of virtual servers. This approach led to the growth and expansion of the Cloud.
In the meantime, containerization was thought of and created, reducing the size and job of virtual machines but actually keeping being some kind of virtualization of an environment, so keeping the server as the model to follow. Platform as a Service was IaaS's heir, not changing the idea at the base.
The paradigm that instead broke the mould is serverless. Up until this point, the various paradigms were just mimicking servers in one way or another: with this approach, the idea of a server and the idea to think to servers disappears. In fact, in particular with Function as a Service, the core becomes exactly the *function*, that is the logic, and not all the boilerplate built around it to make it run.

Understanding this evolution is a key to comprehend the power of serverless: the *caring threshold* moves up, or better, everything that is not strictly related to the business moves down below it.
But it is not enough to fully comprehend what advantages a serverless architecture can provide to your infrastructure: the most efficient way is to prove it by yourself. The patterns

to follow to design a proper serverless application are in some cases new to the classic distributed application world, and all the phases of the creation of an application need to be evaluated again.

These are the main reasons behind the choice to face serverless in this work. The goal is to explore the new patterns, the reasons for their being and their application, what choices are offered, understanding enough to obtain the tools to choose the best fit for each use case.
The real tools to explore this world is offered by various cloud players, each providing a kit characterized by pros and cons, and in particular for this work the chosen provider is Amazon and its Web Services. AWS has been on the market for more than a decade and their services to create a serverless FaaS architecture are the most complete around.

As for the writer, dealing with this topic is a matter of Software Engineering. As such, the choice about this work is not to include any line of code among the main chapters but delegating them to the appendix: the crucial part is not the final realization, that is the code running on the system, but its designing.
Furthermore, this work is structured following a top-down, breadth first approach: one chapter after another almost the same themes are dealt with, changing perspective and getting every time a step closer to the actual implementation.
In the last chapter, in fact, a prototypical application is realized: UBS, a hypothetical yet finalized book store application where users can purchase and sell books. In the creation phases of the prototype, from designing to deployment, several issues get faced up and various patterns followed to solve them.

Before possibly dealing with serverless, though, the general microservices architecture needs to be faced up to. Most patterns, in fact, come from this architecture, in that serverless could theoretically be considered only a deployment style for it.

1 Microservices

1.1 Definition

The term “Microservice Architecture”[1] has risen and spread in the last years to describe a new way of designing, developing and deploying distributed applications. But, while many people around the world talk and write about it, a strong and precise engineering definition still lacks. There are no boundaries on the language to use or the technologies to meet the software’s needs.

A microservices architecture is such when it is built on top of many small independent, interoperable and interchangeable pieces of software, not directly communicating between them and not knowing each other.

The benefit of structuring an architecture like this is that it improves the application’s modularity, thus making the application easier to understand, develop, test, allowing separation of concerns, so that each module can be assigned and developed in parallel, then maintained, each by an independent, small and autonomous team.

Keeping in mind the previously highlighted points, you can say you are looking at a microservices architecture, and you can define an architecture as a microservices one, if each service it is built on meets some requirements:

- it is independently deployable
- it is easy to replace
- it is small
- its area of interest is *bounded*

[1] The term "microservice" was discussed at a workshop of software architects near Venice in May, 2011 to describe what the participants saw as a common architectural style that many of them had been recently exploring. In May 2012, the same group decided on "microservices" as the most appropriate name.

A software built as microservices can be broken down into multiple component services. This approach allows the system to be more robust by letting single services' failure happen and be restored and fixed independently.

Each service deals with a single business capability. This lets the company go full-agile, and instead of having teams made of people divided by their specific skill, building many smaller teams covering all the skills they need to entirely build their service. The basic idea comes from the computer programmer Melvin Conway in 1967 who stated what now is called the Conway's law:

> *"organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations."* (Conway, 1968)

Even though this statement is, at present day, 51 years old and most computer programming paradigms have radically changed, it is still valid, maybe more than ever, on microservices architecture.

Each service is smart in its logic but dumb in the result management. This means that all the service must deal with is the transformation of the input and the production of the output. What happens next is not its business, but another service's business (maybe its only).

So, it is natural to have distributed governance, not only for the process, but for the teams themselves. Each team must be responsible for its own process and should decide by itself what tools are the best for their workflow and this leads to teams producing useful tools for other teams to reuse it, thus to improve deploy time and robustness.

Not only governance is decentralized, but also data management: each team must oversee all the data its process deals with, and this has as result the chance to choose the best system for the specific use.

Finally, independency leads to what maybe is the most propulsor to MSA: scalability. Scaling up an entire monolith is obviously possible, but highly inefficient. With microservices, the easiest approach for scaling up, the duplication can be very cost-effective, because only the most requested services can be duplicated on demand.

1.2 Microservices vs Monolith[2]

Sometimes in science the best definition for something is based on what it is not: dark is absence of light, empty is absence of matter. Though it might be not precisely like this for microservices, thinking to monolithic architecture can help the comprehension.

A monolithic application is a big project made of thousands of components, strongly bound between them, possibly running all together into the same system and deployment package, spanning all the areas of interest of the business process.

It's a nice object, the finished project is very powerful and robust, but with dimensions come cost and time. Though robust, its components are strictly coupled: while this makes development phase faster and easier, this may lead to unpredictable changes during the maintenance phase. Furthermore, when a small bugfix is ready to be released, the entire system must be taken down, updated, and restarted.

1.2.1 Advantages

Comparing Microservices and Monolithic architectures, the advantages of the first are evident. First, the system can be ready much sooner, because each service can be developed in parallel; in fact, *Continuous Delivery* (Jez Humble, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 2010) approach can be naturally applied, so that the entire system can be produced in short cycles. This approach helps in reducing the overall cost and the risk and of each delivery by allowing for more incremental updates to applications in production.

Because of the independency between the services, the changes' result can be predicted more accurately. Moreover, if a change is made to a service, it is the sole component that needs to be updated and delivered, tremendously improving availability and leading to smaller downtimes: deploying a single service is usually a quicker action than doing it on a monolithic application. Also, if a service breaks, the rest of the system can keep on living: the dead service will be ignored, and the system's failure will only be partial.

[2] The term monolith has been in use by the Unix community for some time. It appears in The Art of Unix Programming to describe systems that get too big.

For inter-process communication, the usual choice is to adopt message oriented and human-readable lightweight protocols. This helps the developers and the maintainers in comprehending what is happening into the system.

1.2.2 Disadvantages

This architecture is not flawless. Although very interesting in the design, this is not the only phase in a product's lifecycle and so it is not the only one to be considered. The main downsides can be summarized to these points:

- expensive remote calls
- coarser-grained remote APIs
- complexity and cumbersomeness

Microservices inter-process communication is more expensive because it happens between separate systems. Moreover, rarely microservices communicate directly between them, because they must be kept independent and be loosely coupled, and this is usually done by adding a third party, another system whose only job is to put in touch the two parts.
Even when designing and developing a microservices architecture some difficulties must be kept in consideration, and they mostly deal with bounding contexts and defining each services' duties and rights. Having different developers (teams) taking charge of different microservices, the risk is not following a spread standard and not creating a central know-how.

1.3 Scalability

Scalability is one of the main topics pushing forward microservices despite classic software models. It is defined as the capability of a system to handle a growing amount of work, or how fast it is to accommodate that growth.
Classical scalability that regards Software can be differentiated between horizontal and vertical. To scale horizontally means to add a node to the system equal to the previously available ones. Vertical scaling deals with adding resources to each node.

In microservices scaling as such is achieved by redundancy of each microservice (horizontal scalability) or by increasing compute power of each node (vertical scalability). But in this context, the classical model is not the best fit.

In fact, an interesting model to be studied when talking about scalability in microservices is the *Scale Cube* (Martin L. Abbott, 2009).

This model leaves behind vertical scalability because it is not (always) controllable by the designer, and adds two additional levels, thus having a 3-way scaling:

- X-axis scaling, that matches the horizontal scaling
- Y-axis scaling, that splits the application into multiple, different services, each responsible for the functions related to a context
- Z-axis scaling, that is quite like X-axis, but instead of "randomly" assigning the work to one of the equal nodes, distribute the load by some logic, in most cases dividing data in subsets.

Usually a microservices architecture takes advantage of only Y-axis scaling, but there are situations in which all the three axes of the Scale Cube can be exploited:

- the full service is divided by context in multiple microservices
- each of these microservices has some replications by default and can scale up on Y-axis on demand
- data flow can be partitioned among various data subsets, each of them associated to a group of microservices replicas

1.3.1 X-Axis scaling

This is the simplest form of scaling, where an object gets replicated several times and, through a load balancer, the load is split among them. Only the processing instances are replicated, while the database is a shared one. This makes each instance's cache become very large in low time, and the cost of maintaining them grows rapidly. As the system's complexity grows, efficiency of applications designed in this manner decreases. Moreover, updating the codebase is very time consuming, because the updating must be replicated to all the instances.

Automatisms, though, can be applied to solve these issues and make the delivery faster and more tolerable. In some cases, in fact, this type of scaling is applied to microservices too: each microservice can be replicated at runtime to fulfil increasing load, and as the load diminishes, the replicas are killed.

1.3.2 Y-Axis scaling

The application is subdivided into small pieces of application, having independent logics and having, each of them, an own database. Moving the database into the subsystem itself solves various problems. First, there are not big caches to be maintained. Delivering an update is less time-consuming too, because only a single instance must be updated.

In microservices X-Axis is sometimes combined with Y-Axis, thus producing an architecture having the overall system split into multiple subsystems, each of them existing as one or multiple replicas.

An appropriate solution is in fact to have a central logic receiving many (or all) requests and dispatching them to the correct subsystem. An additional logic can be given to this processor, making it decide whether the replicas being present at that moment are enough to fulfil the incoming requests or more replicas must be instantiated.

1.3.3 Z-Axis scaling

This model is very interesting to be studied but in practice is not very common. It has many similarities to X-Axis scaling with the difference that the database is not a big, shared, one but each instance has its own, dedicated to only a subset of data, making the instance itself focused on only a subset of data. There must be an external processor receiving requests that must have some knowledge on which instance is responsible for the subset of data the request is about to correctly dispatch the request.

1.3.4 From Conway's Law to Bounded Context

As previously mentioned, the key idea of such an architecture and its scalability is the decomposition of the full services in many microservices. There can be defined various concerns around which to make them orbit. With regards to the previously presented

Conway’s Law, one of the most common ways to split up the full service in chunks is by business capability. Martin Fowler, in his Domain Driven Design, defines, among many others, a key Design Pattern in microservices: **Bounded Context** (Evans, 2003).

Let's consider an online store: a user logs in, chooses one or more products he wants to purchase, adds them to his cart, that he maybe had already populated with some other products during a previous visit, then he decides to check-out and do the purchase. As far as it concerns the user, the work is done. Still, on the back side, the order must be processed and eventually a courier is chosen to perform the shipping. The courier will confirm his availability to do the job after having logged in, possibly through a separated portal, confirm the pick-up and then the delivery.
The agency providing this service will most probably be divided in departments as such:

- customer care, to interact with the customers
- product management, that handles the catalogue
- commercial product management, providing the price of the product, that might change over time
- warehouse management, managing all the products' movements
- order management, whose purpose is to follow and process the order
- finance management, that handles income and outcome
- delivery management, to interface the warehouse with the external couriers

Following Conway's Law, we could identify these Bounded Contexts:

- Customer
- Product
- Commercial
- Warehouse
- Order
- Finance
- Delivery

These context, nevertheless, are not enough for correctly structuring the entire architecture. At least a Cart context should be added to manage the carts. If the service should support reviews to products, a Review context shall join the others.
The goals of such an organization are various and can be summarized in these three main ideas:

- let each context focus its efforts on what it can do best
- make any context give its more appropriate meaning to each keyword
- security

An increase in security in fact is a side effect that is obtained through inaccessibility of the resources privately accessed and treated by the other context's resources. Having private resources forces the designers to create well defined public access points and interfaces to both the external world and the other microservices. Moreover, there are other drawbacks that mostly deal with data inefficiency.

1.4 Persistency in microservices

To ensure loose coupling and to maximize each microservice's performance, several databases are created. In fact, even though thinning more and more, differences between each DBMS exist, and to these must be added those between classic RDBMS and NoSQL exist and must be accounted. While a microservice's data interactions are best fit on a PostGreSql database, another can give its best by using a MongoDB database, and another might need to do queries that best perform on graph databases. This is the so-called *polyglot persistence architecture* (Scott Leberknight, 2008)[3]. Being free to use any of these at will is the best situation a designer could dream of.
A problem arises, though: how to maintain data consistency?
In distributed systems data has the potential to become inconsistent with ease, mostly due to the interactions among the systems that might happen during data flows, and sometimes because of the parallel tasks that each system can run. When designing a distributed system these problems must be taken care of. The standard approach to address this issue is to think

[3] http://www.sleberknight.com/blog/sleberkn/entry/polyglot_persistence

to data flows directed to storage as transactions, that is, indivisible operations that can either completely succeed or fail. If saving data to a single entity, achieving this is free. The problem arises when many entities or even many databases are involved, and it is even worse when the databases are spread across several systems.

A quite widely used algorithm addressing this problem is the **two-phase commit protocol** (2PC), a type of commitment protocol that coordinates all the processes participating in a distributed atomic transaction, deciding whether to commit or abort it. To make the recovery in case of failure possible, the participants to the process are asked to log the protocol's states. In the best-case scenario, that is even the most frequent one, the protocol consists of two phases:

- commit-request phase, in which a coordinator invites the participants to take the necessary steps for either committing or aborting and declare the result
- commit-phase, in which, according to the participants' results, the coordinator decides whether to commit (if all the results are successful) or fail (otherwise). Then the participants complete the job.

It has some limits, though: it is not fail proof, and in many cases of failure there is the need of the intervention of a human to restore the correct state of the data. Furthermore, being a synchronous protocol, each *participant* is considered busy until the coordinator frees them by sending a result message. This last characteristic makes it not prone to be used on fast microservices, even if the side effect for them is to lose something on data reliability. Side note: it looks like there is no *native* support for this protocol on modern cloud frameworks and services.

The protocols microservices architecture is more inclined to use are based on the eventual consistency. Data is constantly changing, and a workflow might span several nodes and services. Depending on the depth of the workflow, if any, you can rely on eventual consistency or, in harder and intricate workflows, **Compensating Transactions (CT)**(Paul Robinson, 2013).

Eventual consistency is a model counting on asynchronous jobs: while they are being performed, the overall state of data is inconsistent, but eventually, after all the jobs have been terminated, data make sense.

1.4.1 Saga

Compensating transactions answer to the challenge of building a consistent model even in case of system failures. Simply rolling back the data to how it was at the start of the transaction might not result in correct data, because it could have been changed by other services, or instances of the same ones, running in parallel with the failing ones. CT pattern proposes to build special entities handling the recovery of the states by applying modifications reversing those applied by the completed jobs of the workflow. In practice, though, instead of designing aside of the original workflow a reverse workflow, it should be designed a more complex workflow that considers the failures and has some paths to recover the system starting from any possible point of failure.

Coordinating these compensating transactions by creating a logical workflow is defined by the **saga** pattern (Richardson, Microservices Patterns, 2018). According to the definition, a saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule, then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

To coordinate sagas an orchestrator can be defined to tell the participants what local transactions to execute. Otherwise the coordination can be distributed by using message queues or pub/sub events.

In the previously presented online store example this pattern could be applied to handle the order creation process. This process spans at least two contexts, that is, Order and Customer. If covering the entire purchase process, Shipping service should be added to the system.

Considering the Choreograph-based coordination and the usage of pub/sub pattern to dispatch events, the process could be summarized in this way:

1. Order creates an order in *pending* state and publishes an *OrderCreated* event

2. Customer is subscribed to *OrderCreated* and internally tries to reserve the amount of credit needed for the purchase; then in case of success publishes a *CreditReserved* or a *CreditNotSufficient* event
3. Order receives the event and updates the state of the order to either *approved* or *cancelled*

A coordination like this works and is quite used. There is not any third external dispatcher with built in logic that is responsible for the flow. Still, it needs a strong coordination at design level, and the debugging of such a process might become quite harder than an orchestrator-centred coordination. Moreover, if the events become numerous the entire system can become chaotic.

In an Orchestration-based saga, the process can be designed as this:

1. *Order* creates an order in *pending* state and starts a *CreateOrderSaga*
2. *CreateOrderSaga* sends a *ReserveCredit* command to *Customer*
3. *Customer* internally tries to reserve the amount of credit needed for the purchase and sends a result
4. *CreateOrderSaga* in case of a successful result sends an *OrderApprove* to *Order*, otherwise sends it an *OrderReject*
5. *Order* updates the state of the order to either *approved* or *cancelled*

This process is more straightforward but creates a stronger bond between the services.

Despite the strength of this saga pattern, it has some drawbacks, the most important one being the complexity of the programming model. Each pattern, in fact, must be designed so that in case of failure of the entire transaction, the overall system status goes back to a coherent one, that might not be the one before the transaction started. Moreover, for this transaction to be reliable, the change in the data and the publishing of the event (or call to the subsequent command) **must be atomic**. For this to be possible, some patterns have been studied:

- event sourcing
- database triggers

- transaction log tailing
- application events

1.5 Microservices Patterns

When designing applications or platforms, the problems that the designer faces, yet with their differences and particularities, can be modelled and extracted from the real context, and then be mapped to common problems that in software design literature have already been faced and solved. The general and reusable solution, not finished in the source code or in the design but described in both the problem and the solution, is what is defined Design Pattern. Some remarkable patterns about microservices architecture are next to be defined. Many of them deal with the open issues related to applying the previously described *saga* pattern: database triggers and event sourcing are two best practices to obtain atomicity in the triggering of the saga and the storing of the data that triggered it.

1.5.1 Database triggers

This pattern is bound to *saga* pattern (Richardson, microservices.io). It is the simplest solution to obtain atomicity in the process of storing updated data and publish an event that is related to it. The event is, in fact, the update of the data itself. Depending on the system's implementation, this can be achieved in many ways:

- Amazon's DynamoDB and competitors' equivalent systems let systems be invoked on the update of data
- if the previous option is not available, or if a stronger disjunction between contexts is preferred, a proxy events database can be included in the system
- in the case of the usage of less sophisticated technologies, e.g. using only RDBMS databases, the preferred way should be to adopt an EVENTS table in which data is automatically inserted by triggers on the source tables, and make a process poll this table waiting for events to be published.

1.5.2 Event sourcing

Saga related pattern, it is the main alternative to *database triggers*. This pattern forces the user to change his mindset from a data-oriented to an event-oriented storage (Richardson, microservices.io, s.d.). Its working, in fact, is based on persisting the state of a business entity as a sequence of state-changing events - this solves the saga issue because storing a single unit of data is, by definition, atomic. The complexity is moved to the reconstruction of the entity current (or at a given time) state. To diminish the impact of this drawback, the system can automatically create periodic *snapshots*: the reconstruction of the state is based on all the events that have occurred after the latest snapshot.

The event store also behaves like a message broker, thus when a service saves an event to an event store, it is also delivered to all the subscribers.

It must be underlined that the event store is the only source and destination of data regarding the entity it is responsible for: it is the authoritative data source. In this way, concurrent events cannot cause any conflict because they only represent a change in the state, they do not claim to be bearer of full state information.

The consumers of the event store are either external systems and applications that subscribe to the events produced, or *materialized views* that are as such maintained and that can provide immediate eventual consistent data to those who request data at any moment, e.g. for the presentation layer (cfr. Command Query Responsibility Segregation).

Some considerations must be accounted when designing a system using this pattern, and most of them depend on the concept that event data are immutable: when an event has been published, it cannot be revoked or changed. Binding this pattern with Compensating Transaction pattern is almost mandatory and makes the latter easier to be implemented.

1.5.3 Command Query Responsibility Segregation

This pattern is based on the idea that the reader and the writer might be different processes and use different models (Young, 2010). The mainstream approach for creating and consuming data is CRUD. So, creating, reading, updating and deleting. Even modern approaches for creating APIs are quite CRUD oriented, REST is the main example. As the model becomes more sophisticated, the usage of CRUD becomes insufficient. Data retrieving might

be less direct, and some aggregations could be the actual data needed; when updating data there might be some validation rules that only allow certain combinations of data to be stored, or even data stored could differ from the data the client can furnish the backend. The approach the designer use, anyways, is most of the times to match the data model to the data flows, adding layers over layers. This can become confusing very easily. **CQRS** proposes to separate **Command** and **Query** models, thus avoiding these models to be forced as equal. The difference in these models can either be on the object itself, i.e. in its content, or in the interface to access it.

This pattern gives its best with event-based programming models, e.g. the previously mentioned Event Sourcing.

1.5.4 Transaction log tailing

This pattern is related to the usage of *saga* pattern. The working is quite simple in the idea: sniffing the database's transactions log to make each transaction be published automatically and atomically as an event. This pattern is transparent to the application, thus does not require any changes to the codebase to be applied. Nevertheless, it is quite obscure and DBMS specific: every database has its own transaction log system coding, so a different dispatcher must be written for every database.

1.5.5 Communication among services

Let's now consider the previously discussed *Scale Cube* pattern(AKF Partners, 2017)[4]. When scaling on the X-Axis, thus creating a service's replicas to correctly meet the growing load, the APIs entry-point is constantly changing because the logical or virtual address of the process itself keeps changing. A client cannot know in advance who oversees the meeting its request. If the Z-Axis is applied too, the client must decide what service must deal the data it needs to interact with, so in this case the client should be given some logics that possibly changes over time and that the designer might not want to publicly share.

The Y-Axis case is the easiest to approach in theory, because the services are well defined.

[4] https://akfpartners.com/growth-blog/splitting-applications-or-services-for-scale

Services typically need to call one another. Even though each service is responsible for only its concern, to satisfy the end-user needs they need to communicate with each other.
In monolithic applications they are just simple objects and they can call one another in several ways: maybe not directly because of scope limits but using some design patterns a way can be found.
In a traditional distributed system, services have well defined, and predetermined, address and entry-point, so some protocols can be defined to make remote procedure calls or modern calls through APIs.
A microservice-based application instead runs on a virtual system or containerized environment, having no predefined location. The size of the system changes too and so the available replicas of each service do.
It needs to be defined a way to make the services know each other and make the API Gateway correctly route incoming requests. This way is to have a single Service Registry that knows and keeps updated this information (Richardson, microservices.io, s.d.).

1.5.6 Service Registry

Each service instance must be registered at start-up and removed at shutdown. If it has some problems and it is no more available to perform its duty, the Service Registry must know it. Being so important, the Service Registry must be highly available and very robust. And obviously it must work on a fixed, well-known location.
To fill and to update the Service Registry appropriately, thus adding a service's rising instance or removing a dying one, it needs to be defined a model. They are defined two ways for doing this job: either the instance declares itself or a 3rd party registers the instance.

In case of *self-registration,* the service on start-up registers itself, declaring its entry-point characteristics, and ensures it is discoverable. It is its duty to periodically renew its registration too, to make the registry know it is still alive and it can correctly perform its duty: this lets the registry decide that in case of missing periodic check-in the service's instance might have incurred into some problems and must be removed from the registry (until a possible delayed renewal). Just before shutdown, the service will have to declare it too and so unregister itself

from the registry. This approach is simple but needs the designer to implement it. Another drawback is the tight coupling it causes between the service and the registry.

To solve these issues, it can be considered to use the 3rd party registration. In this model, there is an object that runs along with the instance's service, declaring the start-up and the shutdown, and perform more effective health-checks on the service. It has some drawbacks too: it is yet another component to be installed and maintained, making the overall system's structure even more complicated.

1.5.7 Service discovery

To make a client interact with the correct service the Service Registry should be used because, as mentioned, it is the only reliable component that knows exactly the status of the services' instances.

A crossroads rises: either the Service Registry is available for the clients to be queries or it is in an area where the access is restricted: it must be decided whether to put it into a Demilitarized Zone or in a restricted area. According to this decision, a pattern to do the service discovery can be applied to the system.

There are two models:

- client-side service discovery
- server-side service discovery

If the access to the Service Registry is publicly allowed, the client-side service discovery pattern must be applied. The client queries the Service Registry to obtain a service's location, maybe caches the result, and queries the location for the service to be performed.

N.B.: having the Service Registry public doesn't mean that the registration APIs are public. It can be configured to make only requests for registration/dropping available only if coming from specific addresses ranges.

This approach has the main drawback to tightly couple the client to the Service Registry, whose location is fixed and predefined. Changing a Service Registry's location forces the client to be updated some-ways and it must be done for every available clients.

In this pattern there is not any component that is specialized in performing some load-balancing: the only available one is the Service Registry, that must be chosen or designed appropriately.

To avoid these drawbacks, or if the designer chooses not to expose the Service Registry for hiding some information, the server-side service discovery pattern can be applied. All the requests must be directed to a single, fixed and predefined, additional component that acts as a Router to the requests:

1. receive an incoming request
2. analyse it to understand what service must deal with it
3. query the Service Registry (or a local cache) to identify the service's location, performing some load-balancing
4. route back the response to the requester

It is evident that using this model there are more hops between the request and the response, so it might be even slower than the client-side version. And like in all the other cases where a component is added, the designer must be aware of the issues that come with the insertion of the router. The router must support the necessary communication protocols to talk with the clients.
Aside from these issues, there are several reasons why to prefer server-side over client-side, one of them being the simplicity in the implementation, due to plug-and-play components and the separation between client and services.

1.5.8 API Gateway

Each service holds its part of information and is accountable for it. But a client, on the other side of the fence, needs data that comes from several contexts at a time. A client will hardly need only data coming from a context but will need data related to them, coming from several other contexts.
Keeping in mind the leading example, a customer visiting a web page displaying a product of the online store, so using the web client, will see cross cutting concerns information:

- some information about the product itself, such as the name, some descriptive text
- available purchase options
- overall average rating
- some catching reviews
- information on the seller
- average rating of the seller

But the services are organized around Bounded Contexts. Information about a service's concern cannot cross the service's boundaries to be fetched by another service's API. This means that the granularity of the APIs publicly provided differs from the one of the actual total APIs. In fact, it is preferable to avoid having the clients sending several requests to obtain the data about a sole use case, if it is possible. Renouncing to this model might be a better solution, in some cases: if a client needs to be constantly fetching a single domain's data to have real time information, some separation should be considered.

The pattern to solve these problems is the **API Gateway**, that is, an interface interposed between the services APIs and the clients. Its main job is to fetch and adapt the data to fit the clients' needs, to hide confidential information and to deal with security too. In fact, being the single entry-point to the system, it's best suited to block or consent queries.
On the public side, the API Gateway shows some APIs that allow the clients to retrieve all the data they need without having deep knowledge of how the data is organized. In this way, the data or information partitioning can be easily changed without making any single change on the clients: only the API Gateway will have to be modified. Moreover, there is no need to share a protocol from the client to the final service.
To this element of the system it is often given the role of API composer: *API Composition* is a pattern in which data provided by different services can be joined through in-memory processing to provide a single appropriate result.
Finally, the API Gateway is, or is the responsible to communicate with, an entity called Service Registry that is the connoisseur of the internal APIs spread across the services.

Keeping it simple, the API Gateway is a component that acts as Router (cfr. Service Discovery), Service Registry and most of the times as events dispatcher too.

1.6 Deployment of Microservice

In the previous sections the focus was on the general microservices architecture, how to make microservices interact with each-other, how data should be structured and what patterns have been studied to solve all the issues that come with the separation of concerns among separate and independent entities.

What has not been yet talked about is how a microservice can, and should, be implemented, how it is structured, deployed and what patterns exist focusing on it.

Basically, a service will run in on a virtual machine or a container containing all the microservice structure will be defined and deployed on command. A service must be provided with the appropriate CPU, memory and I/O resources. Each service can be written in a variety of languages and moreover each of them can have its own specific deployment, scaling and monitoring requirements. Despite this complexity, deployment must be fast, reliable and cost-effective and the behaviour of each service's instance needs to be monitored.

1.6.1 Multiple service instance per host

On an architecture structured by using this technique, virtual hosts are shared among diverse services' instances. Because of the usage of well-defined entry-point, it is difficult to make many instances of the same service share the same host. This pattern is still useful, though, because it allows to make deployable suits of services' instances. For example, two hosts could be created to run the same services, one instance per host per service.

About JVM or Node.js services, the most common solutions, each service instance can either be deployed on a separate process or on a common one. The two cases share the same pattern but have some differences. In either case, the great advantage of this pattern is the efficient resources usage it comes with: multiple services instances share the same host and operating system. This means low overhead, even lower in the case when multiple instances share the same process too. Another benefit of this pattern is the deployment ease: just copy the service

and run it. Because of the low overhead, starting a service is not so impactful on the performance calculation.

Nevertheless, this pattern has some drawbacks. One major one is the almost total lack of isolation between services instances: when sharing the same process, like in the second type of approach where multiple services instances not only share the same host but the same process too, there is no isolation and so, for example, they share the same JVM heap; a misbehaving service could break the other services it shares the process with. In the same scenario, it becomes very difficult to accurately monitor each service instance's resource utilization, and it cannot be finely tuned or limited too. A misbehaving service could consume all the resources dedicated to all the services running in its process.

1.6.2 Service instance per host

Deploying services following this technique leads to a total isolation among them, having each service run on a dedicated host. What host means in such a situation is either a Virtual Machine or a Container, so two sub-paths can be followed.

Choosing a path where a service instance is hosted per Virtual Machine, it is created a virtual machine image for each service, and thus an instance is a virtual machine that is launched using that specific image. Such an image contains not only all that is needed to run the service, but also the operating system on which the service runs. The major benefit of this approach is the complete isolation it is obtained: it has a fixed amount of CPU and memory and cannot interact with other services or steal their resources, either correctly or wrongly behaving.

The overhead and the slowness in the delivery, though, are the main disadvantages to this pattern. Building a new virtual machine image when a latest version of the service needs to be deployed is hugely impactful on the delivery time. When it comes to running cost, it becomes clear that this approach is not resource efficient. In fact, each service instance has the overhead of the virtual machine, considering the entire operating system it runs on. Moreover, in the typical Infrastructure-as-a-service where the services are deployed on nowadays run fixed size VMs, so the service's virtual machine could be underutilized. Being the cold-start of a virtual machine very slow, it is not prone to be created and destroyed

frequently, so it is preferable to run a slowly changing number of instances at a time. This means that most of the time the resources payed for do not fit the need of the service.
Finally, there is no provision for the managing of the virtual machine, and this adds complexity and cost to the designer.
As an alternative to this technique, due to the several disadvantages it comes with, an alternative in the entity to make the service run on has been defined: instead of dedicating an entire virtual machine to a service's instance, a container is enough to guarantee isolation and improve performances.
Following the technique named *Service instance per Container*, in fact, each service instance run in its own container, that is, a virtualization mechanism at the operating system level, consisting of one or more processes running in a sandbox.
It has the same advantages as when running a service on a Virtual Machine, so the total isolation from each other, the ease in monitoring the resources consumed, the encapsulation of the technologies used by the service. Moreover, this approach solves the major issue of the preceding one, that is the extreme overhead: creating, deploying and launching a container is much quicker, and this solves the cost related issues previously discussed. This approach has some drawbacks too, mostly related to the decrease in security with respect to virtual machines. Also, unless using a provisioned container service, the containers infrastructure must be managed, and possibly the virtual machines infrastructure it runs on too.

There is a final solution to solve these issues too, and it is the Serverless (Ken Fromm, 2012)[5] Deployment.

1.6.3 Serverless Deployment

This is the most modern and appealing way to deploy a service. In fact, it solves all the disadvantages of the previous pattern, simply because the infrastructure the services run on are not to be managed but are provisioned. This means that (almost) only the code the service runs to execute its tasks has to be taken care of.
There is some drawback too, such as the time-limit and the stateless they come with.

[5] https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/

2 Serverless

When talking about microservices deployment technologies, a kind of zooming in has been done. The first considerable technology is what now is called monolith. Here, there are some services that are built to be deployed and run altogether into a “huge” system. Each service has its dedicated processes, technologies and package. They can share some resources, e.g. databases, but they are independent on all the rest. They can call each other by using several technologies and patterns that have been defined over the last decades. Then, the first ideas to separate processes and spread them over several systems have been thought, and distributed applications are born. Distributed technologies have moved from client/server approaches to multi-server, and the first microservices patterns arose. Within this architecture, microservices stopped sharing the same physical host and started sharing at first virtual hosts directly, then by passing through containers. In this way, a system can consist of several physical machines each running several virtual machines each orchestrating several containers. Each container, usually, only runs a single service’s instance. In this way, a multi-layer structure gets clearer and clearer:

1. physical layer
2. operating system layer
3. virtual machine layer (i.e., another operating system layer)
4. container layer
5. code

In this last approach, the service’ manager has to take care of the entire stack. Or at least, it was like this. With this approach spreading more and more, the height of the stack that must be maintained got lower and lower:

1. the first 2 layers get abandoned altogether by hosting the services on remote virtual hosts, so that only remote virtual machines must be taken care of, installing all the tools and the technologies needed to run the services

2. then it's time for abandoning the virtual machine layer, by using remote technologies that allow the service owner to only worry about the creation of a container that encapsulates all the technologies the service needs to perform its duty

What happened is that the complexity of the system gets moved from the end-user to another entity. This entity simply is the owner of another service: distributed technologies become distributed services, and the previously mentioned stack of layers needed to run a service becomes a stack of services to run the service onto.

The next obvious step is to abandon the creation of the container: most of the times the containers only differ for the code they will run after deployed. The next step is for the service owner to only take care of the code, the logics that are built around the business.

2.1 Serverless computing

In the introduction some of the keywords of the serverless computing model have been suggested. In this model there is an actor, the **cloud provider**, that dynamically manages the allocation of resources. With this approach, the service owner writes the code and deploy it to the provider, tunes some settings, and then the provider takes care of scaling to provision enough power to meet the load. This approach is now called **Function as a Service** (*FaaS*). With this architecture, the remote services execute on stateless containers deployed and run when some events occur, that die right after and that are fully managed by a third party.

In recent times, with the huge diffusion of smartphone and the recent mobile web usage overtaking the desktop one, the need of having client application smart enough not to have a server behind the data processing caused the birth of systems where, through APIs or dedicated SDKs, clients were auto sufficient in data-processing. This approach is now defined as **Backend as a Service** (*BaaS*). There is no code running on remote, only on clients.

It is not required that the entire application is built serverless, in fact often only some specific jobs are assigned to *functions* in case of FaaS or performed by clients on BaaS. If the application is somehow hybrid, it is best combined with services running as microservices, because these two types of architecture share most of the patterns.

2.2 Function as a Service

AWS Lambda must be mentioned right away because most of the merit for the diffusion this architecture is reaching is due to it. It is the *Amazon Web Services* provisioned solution and the most used one. In the Lambda introduction page, it is stated that

> *AWS Lambda lets you run code without provisioning or managing servers. You pay only for the compute time you consume - there is no charge when your code is not running. With Lambda, you can run code for virtually any type of application or backend service - all with zero administration. Just upload your code and Lambda takes care of everything required to run and scale your code with high availability. You can set up your code to automatically trigger from other AWS services or call it directly from any web or mobile app.*
> (Amazon AWS)

Leaving aside the marketing and the commercial sentences, there are some points that should be underlined. First, it is highlighted the absence of long-lived server applications, that forces the application to be stateless; then the possibility to choose (almost) freely the technologies to adopt; then, the total difference in the way of deploying the application with respect to monoliths or microservices; horizontal scalability is automatic and only needs to be tuned to meet the requirements; last, the ways to interact with the services, to make them execute, is specified and it is based on events or on direct remote calls.

2.2.1 Components of a serverless architecture

In a *FaaS* architecture, functions should not be called directly, they cannot. Their execution, in fact, is *triggered* by some events, that mostly depend on the *FaaS* platform itself, although some standard patterns are established.

In a typical *FaaS* architecture, an *API Gateway* can be found to map calls coming from the external world, i.e. the clients or other services, to specific functions. The jobs these functions perform are often dependant on streams of data, real time information the system must quickly react to, thus the need of a Message Queue. Talking about messages, the handling of

a *Pub/Sub* communication could be either assigned to the Message Queue or to a specific component dedicated to it.

2.2.2 Ports and Adapters

Before discussing the various components composing a *FaaS* service's architecture, it needs to be clarified the general pattern these functions are organized around. As already mentioned, functions should not be publicly accessible but restricted, thus obtaining control for monitoring, abstraction of security from the implementation of the business logics, interoperability and independence from the communication technologies. It is as such created a kernel of pure business logic.

The rest of the work for letting the world using the logic is responsibility of an external layer that wraps the kernel. Thinking it in this way it is easier to obtain an onion-like structure instead of a classic stack-like one. Lately, in fact, the most discussed patterns for building APIs focus on the construction of layers in this fashion: while *Layered architecture* and *Onion architecture* (Palermo, 2008) are in practice almost the same, the difference is in the way the designer approaches to it. The key difference, in fact, is that thinking the architecture as layered leads the designer to think the layers are dependant and somehow binds them. Just making the layers circle instead of flat and wrapping one into another, though, is not sufficient to obtain a correct Onion architecture. The keyword, here, is *Dependency Inversion Principle* (Martin, 1996).

DIP does not introduce nothing new to classic Design Patterns, it just combines and applies them to an architecture built on layers. It states that:

1. *High-level modules should not depend upon low level modules. Both should depend upon abstractions.*
2. *Abstractions should not depend upon details. Details should depend upon abstractions.*

Plainly speaking, interfaces have been interposed between layers for communication. These interfaces should be thought appropriately, that is, building them independent from the final implementation.

Onion architecture applies DIP to the overall architecture, and defines the standard layers needed for some APIs, listed from the core to the external:

- Domain Model, where all the business logic resides; this layer is agnostic in communication
- Application services, where application specific logic resides
- Outer layer, consisting in anything that directly interacts with APIs, such as User Interface or tests

Starting from an architecture subdivided as this, Alistair Cockburn arrived to identify a more structured architecture that, at first, he called *Hexagon Architecture*, and later refined and renamed it *Ports and Adapters* (Cockburn, Ports and Adapters, 2005).

Hexagon architecture consists of an external hexagon where the outer world resides, that is, in this case, both user side components needed to make the application reachable and usable, and database interfaces, and an internal hexagon consisting of the application core. The news, here, is that the border of the internal hexagon can be considered a layer itself: it is where the external and internal hexagons components meet each-other and it works as a ports layer. The fact that this architecture is represented by using hexagons does not mean that the number 6 is important in any way, and for the same reason the name of the pattern changed to Ports and Adapters.

Fully combining the Onion architecture with the Ports and Adapters pattern leads to a structure where, on one side of the ports, in the inner side of the hexagon, stands an onion defined as before, so consisting, from the inside to the outside, of a Domain model, Domain Services and finally Application services that communicate, through the ports, with the external Adapters.

The adapters could be defined as either *Driving*, that is, those that tell the application to do something, or *Driven*, that are told by the application to do something. User side adapters belong to the first category; internal communication components, persistency objects and systems that let the application communicate information to the users belong to the latter.

FaaS functions obviously belong to the core of this architecture, but the components they communicate with are in fact adapters on the external side.

2.2.3 API Gateway

This is probably the main adapter among the Driving ones, because it creates and publishes public entry-point that allow the various clients (or external services) to use the application. Being an adapter, its job is to map the input it receives to an input that is comprehensible to the destination, that must be kept as much communication agnostic as possible. It is also the main responsible for the Authorization and the Authentication: part of the APIs it shows might be not available to unauthenticated user or to those that are not part of a restricted group (authorization), and part of them should be personalized, in the mapping phase, to refer to the user itself (authentication).

2.2.4 Pub/Sub messaging

More than a component this is a pattern. It consists of 2 direct actor categories, i.e. the *publishers* and the *subscribers*, and a transparent actor, the *broker*. In this messaging pattern, the senders of the messages do not program to send the message they create directly to the consumers, but instead categorize the messages into classes. In the same way, subscribers do not know in advance all the characteristics of the messages they will receive but they know that the messages are about the categories they are interested into.

What is happening when a message is created and published is that the publisher sends it to a central actor, the broker, who is responsible for taking the subscriptions and dispatch the messages; the broker, then, dispatches the messages by applying the filters the subscribers communicated along with the subscription. These filters can be either **topic-based**, if there are several named logic channels where the messages flow, or **content-based**, if the content matches some constraints.

Following this pattern leads to loose coupling because the subscribers do not need to know who the producer is, and the producer only publishes data, without being interested if anyone is interested in consuming it. The topology of the components is not interesting for anybody, the focus is on the messages themselves and their structure, that of course must be well defined and as much immutable as possible.

But with loose coupling some issues come, most of them due to the inherent lack of acknowledge systems between producers and consumers. Consumers might fail or be not operative when an important data is produced and published. Some problems with security

arise too, because there might be published unauthorized messages, unauthorized receivers could spoof private ones, or the broker could be overwhelmed by malicious data flows.
In a FaaS serverless environment, this pattern is generally one that enables the triggering of a function: it is subscribed to some type of messages, and when one is published a new instance of the function gets deployed and started, with the task to treat the message as input.

2.2.5 Message Queue

This is both a pattern and an adapter implementing it. It can be considered a relative to the previously analysed pub/sub pattern, and in fact they are quite coupled. Where these patterns differ is in one of the weak points of the pub/sub: while the latter can cause some inconsistencies when there are no active subscribers, message queue instead stores the messages not yet consumed. In fact, all the messages produced are temporarily stored in a queue, could possibly be sorted and analysed, and then removed only after there is an acknowledgment and a confirmation of the job completed by the producer.
This pattern helps with services where there are continuous flows of data and a real time analysis must be performed.

2.3 FaaS Architecture

Summing up, *FaaS* is a type of serverless architecture, and architecture can be considered just a way to deploy a service organized following microservices architecture. This leads to making FaaS follow the general microservices' patterns and the way of organizing the overall system's structure, apart from the deployment and the lowest level parts.
So, the centrality of Bounded Context is valid, communication between services is still problematic and the ways to invoke services are almost the same: these patterns are shared with microservices, but the way they are applied are different.

2.3.1 Bounded Context in FaaS

When discussing the most proper way to separate concerns and organize data models in microservices, the Bounded Context pattern was highlighted as the most used pattern. The main idea, again, is to build small groups of entities orbiting around a unique business entity.

For microservices it can be said that it is quite easy to obtain such a separation, and various patterns, mostly regarding the deployment, can make it very straightforward to achieve this goal.

For functions it is a little harder: there is not any container that can isolate groups of functions. Alternative strategies must be applied, and they mostly depend on the platform the functions are running on. In fact, it does not exist a defined standard for commercial products that host functions, and it is not thinkable to leave the designer make its own implementation of a platform the functions run on, because most of the advantages in this architecture would be lost.

Actually, an **Apache** project exists, **OpenWhisk**, that allows anybody to create a platform on which to run *FaaS* functions, and in fact **IBM**, sells to end-user a service, *IBM Cloud Functions*, built on it. Anyway, OpenWhisk does not indicate a way to separate groups of functions to build closed concerns, thus following Bounded Context.

The truth is, this pattern does not deal specifically with security. It does not state that contexts must not access other domains' data. Its focus is on the separation of concerns for what regards the logic, the naming, the model. It is a pattern that first must be applied on the design phase, and possibly in the technological implementation.

The way that is currently used on most of the commercial solutions hosting functions to achieve a *physical* separation among domains is through authorization. For a totally provisioned service, for example, where each component is one of the services available on the platform, some roles and rights must be defined and assigned to each function: in this way, functions cannot access other domain's persistency objects.

2.3.2 Communication within the system

To meet its duties, the system needs its component to communicate between them somehow. As already mentioned, the key pattern most of the communications should follow is Pub/Sub messaging: designing the functions to invoke one-another does not lead to anything good. In fact, one of the benefits FaaS architecture leads to is the ease to substitute components to meet the same requirements or update them. When discussing about microservices, it has been pointed out how they needed to be kept uncoupled by not linking them together because of moving, non-fixed endpoints. This is still valid, partially: as long as the function

taking care of a task does not change, the entry-point stands still. Apart from this situation, the reasons why loose coupling between functions must be applied do not change.

Designing communication between functions consists on applying the saga pattern. Both the Choreograph-based and the Orchestration-based patterns are valid in FaaS too, and the usage of one or another is situational.

Choreograph-based coordination comes free in terms of components to be created, applying Pub/Sub pattern.

Orchestration-based coordination, on the other hand, solves the problems the spread choreography comes with, that is, the tremendous quantity of events that flow. Applying this pattern, a third element must be created and deploy along with the functions that use it. Various solutions can be thought, and they differ on the level of specialization:

- a single Events-Dispatcher, that lives outside of any Bounded Context and that handles all the events that deal with direct function to function calls
- dedicated Events-Dispatchers, that lives outside of any Bounded Context, whose only job is to one specific function invoke another
- provisioned services

Both single and dedicated events dispatchers can be created in various ways. An interesting one is by using a FaaS function. Differently to the functions this one should link, that cannot fall out of the Bounded Context they live in, it is given special powers to cross over domains. In case of single dispatcher, it must be built rock-solid to prevent any security-related issues and not to make the whole system stop working. It must know how to translate a message coming from a function, that is, what function must be called and how the input should be translated, if needed. Then, of course, it must check if the caller can invoke the called. As the system increases, this component can become quite huge and unmaintainable.

On the opposite, dedicated events dispatchers are built on purpose to serve a single caller-called couple. There is not any dictionary that must be consulted in order to individuate the destination. Only input translation, if any, and security check must be performed. Its obvious downside is in maintenance because of the quantity of different components.

The alternative is to use provisioned services. They come with the platform hosting the functions and at present day the only commercial solution is **AWS Step Functions** (more on this later).

2.4 Commercial solutions

A serverless *FaaS* architecture needs a system to deploy functions on, and most of the advantages this solution comes with deal with the ease of the deployment and the low maintenance. As such, the most used, and the only for some time, is to use *Systems as a Service (SaaS)* platforms that allow to mostly only write code and deploy it.

Before discussing the commercial solutions, a mention to the **Apache opensource OpenWhisk** project must be done.

2.4.1 OpenWhisk

This project is a serverless event-based programming service that makes FaaS programming deployable on hosts managed by the programmer. In fact, containers can be built and deployed on clusters. As an alternative, **IBM** sells a SaaS hosting FaaS running on this project. In either case, by using the proprietary CLI, functions, named *actions*, can be deployed.

2.4.2 Amazon Web Services

Amazon Web Services (AWS) is a collection of services offered by **Amazon** composing a pay-per-usage platform.

AWS Lambda is the first service for hosting functions in commerce, being born in 2014, November. Being part of AWS, it can be integrated with almost all the other services in the platform. Integrating means that a function can both use other services (e.g. save a file on S3) or be triggered by some events happening on other services (e.g. when a file is loaded to a S3 bucket).

2.4.3 Google Cloud Platform

This is probably the platform that changes the most day by day. The service to host functions, **Google Cloud Services**, was born in March of 2017 and released as a beta, before going public on October of the same year and being apparently merged with Firebase Cloud Functions right after. The biggest issue with this platform is that only Node.js is supported as a language to write functions.

It supports as event sources Pub/Sub for messages, Cloud Storage for data storage related events, HTTP for web requests, Stackdriver Logging for events based on logs and finally Firebase.

2.4.3.1 Firebase

This is the most widespread *mBaaS* system on commerce, and it is continuously improving and evolving. It offers an advanced SDK for mobile and web platforms, that allow the final user to directly interact in total security and efficiency with remote data, in real time. In fact, the strength of *mBaaS* architecture is the clients can hook to some data and listen to all changes happening to them.

Firebase also can host *functions*, to run code that should not or cannot run on end-user devices. These functions can be triggered by changes happening on data, on files or via HTTP.

2.4.4 Microsoft Azure

Azure is the name of the cloud solutions offered by Microsoft, offer that is quite wider in services than Google but not as close to Amazon. **Azure functions** is the service that hosts *FaaS* and it supports various languages.

2.4.5 Considerations

Being able to execute functions in response to events is only as useful as what you can actually do within the execution pipelines, that is, how many and how useful services around functions are. Functions themselves, especially if written on a commonly supported language, do not differ substantially according to the service they are host on. Some minor change can be present on the costs and the latency, but availability and reliability are very high in every cases.

As such, a framework among other, **serverless**, has been created that hides the platform to the developer, letting him ignore the lower level configuration.

2.4.6 Serverless framework

This framework is an abstraction layer on the *FaaS*. It lets the developer write the code of the functions without worrying about the platform-specific configuration needed. As such, it allows the developer to move the function from a service to another, at will (if there is any reason why to perform such a task).

3 AWS Serverless

This chapter will focus on the services hosted on Amazon Web Services that allow the developer to design a full Serverless architecture. In particular, *FaaS* will be explored, but some interesting cues on *BaaS* will not be left behind.
First, the various services will be described, then it will be shown possible solutions to various difficulties that are encountered when creating a *FaaS* architecture based on these services.

3.1 Services

3.1.1 Identity and Access Management (IAM)

This service is the responsible for authenticating and authorizing people and services all over AWS cloud services. It grants fine-grained access control by giving the possibility to define access by user, by group and by permission, to computing, storage, databases and services. It can manage classic authentication, as well as multi-factor authentication, IP filtering and even specific conditions like time of day can be used to control access. It can be incorporated with federated access or existing identity systems.

3.1.2 API Gateway

This component is the AWS service implementing the API Gateway microservices' component, being the backbone on which the connectors between private o public connection engage to communicate with AWS services. It provides RESTful APIs that can be configured to access AWS services.
Behind the API entry points, API Gateway encapsulates access to services by method (API-side) and integration (service-side) requests and responses, that is, by interposing transformation that apply to the services' input and output.

Providing the configurability of RESTful APIs, the API Gateway is organized in nested resources, each exposing standard HTTP methods entry points; each HTTP endpoint can be set an Authorizer, that is, an entity that checks authorization to the method. Authorizer can be either Cognito or a Lambda function.

The mapping between the end-point and the Lambda function or service handling the request can either be explicitly defined or left as proxy, that is, passing all the information about the HTTP request itself. For all the integrations not acting like a Proxy, a mapping template can be defined. AWS API Gateway service template supports **JSONPath** expressions and is written using **Apache Velocity Template Language** (VTL), and thus interpreted by Velocity, a java-based template engine that allows to reference and manipulate objects.

3.1.3 Cognito

Though it can be considered closer to *mBaaS* than to *FaaS*, this service can be quite convenient to use to provide end-user authentication and authorization to the service. In fact, it can be used in combination with API Gateway to control access to the entry points of the service.

Along with these uses, Cognito offers some private space combined with each user, where to store private configuration or data that the user rarely modifies.

3.1.4 DynamoDB

This is the service that offers the main Database solution inside AWS. It is a highly available, autoscaling nonrelational database service that offers super low latency. Being a nonrelational database, it offers a flexible data model, supporting both document and key-value store model. Access to data happens through a specific SDK calls: the flexibility in queries is not even close to that of a relational database, and so the data model should be appropriately studied. Data is identified by using a single partition key or a couple of keys, the second of which being the sort key.

Additionally, DynamoDB Tables support secondary Indexes. Their creation grants better performances for queries where accessing data by primary key is not available. An index can be either local, if it points to subsets of data previously filtered by the partition key, or global, if it applies to all the data. An index can be made of a hash key and a range key, whose purpose

matches that of the couple partition key - sort key. Maintaining indexes has an additional cost that does not depend on the size of the table but on the provision granted to its usage.
Clearly, also DynamoDB integrates with AWS IAM for fine-grained access control of users.
DynamoDB also integrates with AWS Lambda to provide triggers, that enable the automatic invocation of functions. It can be set AWS Lambda to be invoked when some data is written on DynamoDB tables: since DynamoDB offers very low latency in writing, it can happen that in very low time various rows are written, so AWS Lambda allows to set a maximum number of rows to be handled per each function invocation, or just one.

3.1.5 Kinesis

This service offers a data stream where to publish continuous data coming from the external world and needing to be transformed, analysed or stored by AWS Services.

3.1.6 Simple Cloud Storage Service (S3)

S3 is an object storage built to store and retrieve any amount of data from anywhere. It provides comprehensive security capabilities supporting various forms of encryption and protecting data via IAM. It offers Big Data analytics in place, the possibility to perform SQL-like queries to access unstructured data.
It is one of the available sources of events for AWS Lambda, that can respond to changes to the data stored.

3.1.7 CloudFormation

This service provides a common language to describe and provision all the resources being part of an infrastructure in a cloud environment. This means that by composing a single text file containing the structure of a service built with various AWS services, it can be automated the creation of an entire stack of resources, spanning many regions and accounts, if needed. Storing the file, for example on S3, and configuring CloudFormation to do it, the system can be queried through some events to build and deploy the resources contained in the configuration. If some error occurs, CloudFormation takes care of automatically rolling back.

3.1.8 Simple Notification Service (SNS)

SNS is AWS's service that is the main responsible for dispatching messages to various subscribers around the system, that is, both handling Pub/Sub messaging among services and delivering notifications to end-users. It is organized around topics and supports various subscribers, like Lambda Functions and HTTP end-points, and some sources both automatic on events (e.g. S3) and via a proprietary SDK.

3.1.9 Step Functions

This service helps in organizing invocations of Lambda functions by allowing the creation of finite state machines. A classic finite state machine can express an algorithm as a number of states and their relationships, and their input and output. This service allows the coordination of individual tasks by expressing the workflow as a finite state machine. Each state can make decision based on their input, perform actions, and pass output to other states. States can perform a variety of jobs based on their type:

- Perform a task
- Make a choice between branches of execution
- Stop an execution with a failure or success
- Pass data possibly injecting predefined values
- Provide a delay
- Begin parallel branches of execution

3.1.10 AppSync

This service deals with accessing resources and functions by interposing between the client requests and the final functions a gateway. This service can in fact be almost thought to as an alternative API Gateway:

- It is a middleman between frontend and backend
- It hides the backend resources to the frontend adding an abstraction layer
- It can transform data flowing through it
- It can handle security

In fact, it can be used in alternative to AWS API Gateway, or in conjunction with it. To better define what AppSync is, GraphQL must be defined first, in fact, AppSync is an AWS provisioned version of GraphQL.

3.1.10.1 GraphQL

Born inside Facebook in 2012, it was publicly released with Open Source license on GitHub in 2015. It is a data query language, providing an alternative to RESTful API: it allows clients to define the structure of the data required and exactly the same structure of data is returned by the server. In this way, useless excessively large amount of data flows can be avoided, lightening the load on the network and reducing the data consumed on mobile plans.
The end-point provides various queries, for retrieving data, and some mutations, for writing data, the clients can perform and their schemas. Queries will provide structured data: to each composite node of the structure, a resolver can be associated to provide the required data. When a request to some structured data arrives, resolvers are queried in depth, and the data is combined. If a resolver returns some data the client didn't request, GraphQL filters them out.
REST APIs are structured around nested resources (the parts of the address) and the actions that can be performed on each resource are limited to the CRUD ones, mapped to HTTP method:

- POST means Create
- GET means Read
- PUT and PATCH mean Update with some differences between them
- DELETE means Delete

GraphQL, instead, proposes two actions mapped to Query (to read) and Mutation (to write) but there is not any obligation into this separation: a resolver mapped to an object in a query can be correctly used to write data. Resources are represented by structures that can be nested to build sophisticated graphs.
Actually, apart from query and mutation, a third type (not interchangeable) exists, that is, subscription: this should be used to make the client subscribe for some kind of events, mostly by using a Pub/Sub channel.

The main advantage in using GraphQL instead of REST APIs is the tremendous decreasing in the number of requests to the APIs is made to retrieve all the data needed for a single use-case. As an example, let's consider the leading one. To obtain all the information needed by the client to show a Product page, it might need to perform these queries:

- to Product resource, to get the product's specific information
- to Commercial resource, to get the price of the product
- to Customer resource, to check if and when the current user bought the product
- to Warehouse resource, to check the availability of the product
- to Reviews resource, to know the average rating of the product
- to Reviews resource again, to get some reviews

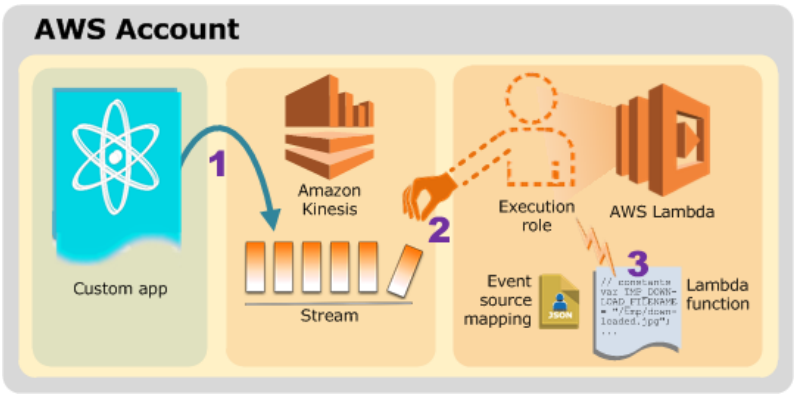
With RESTful APIs the client will need to perform at least 6 queries to only show some information on a single product; with GraphQL only 1 query will be launched by the client, breaking down the latency.

3.1.11 AWS Lambda Function invocation types

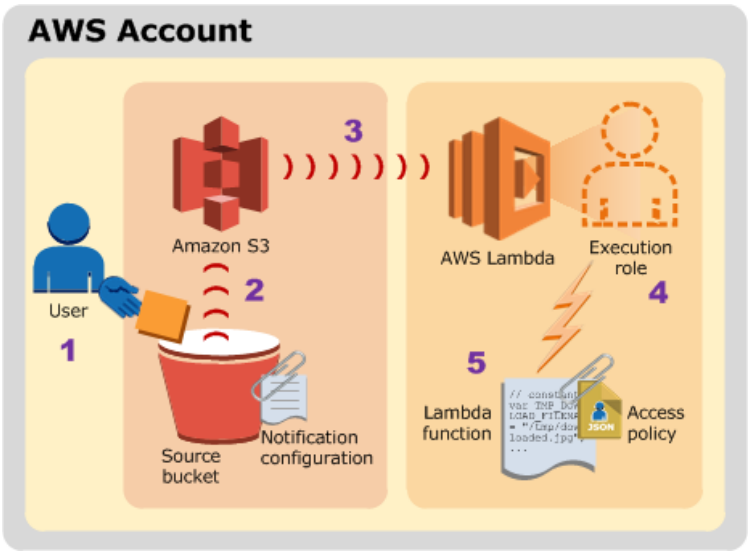
When building a serverless application on the AWS platform, the core components performing actions are the Lambda functions, while *event sources* are the AWS services and the custom applications that publish events for the Lambda functions to process. In addition to the AWS services that Lambda support as event sources, custom ones can be defined by exploiting the AWS Lambda *invoke* operation. These custom sources are decided by the designer whether to trigger synchronous or asynchronous invocations. For event sources based on AWS services, there is no choice left to the designer: it depends on the specific event source (there can be many) the AWS service provides.

An additional classification can be followed to distinguish AWS Services event sources. In fact, they can be based on streams or single events, whether the service can provide a continuous stream of data and it should not be the responsible of invoking functions, or there is not a strong logical binding in the successive events the service fires or if it can be indicated as the responsible for launching invocations.

At present day, only Kinesis and DynamoDB can generate streams and can thus be set as stream-based event sources for Lambda functions. Choosing a stream as a source for events, an event source mapping residing in the Lambda service must be configured so that, at a fixed time rate, the service polls the stream to check whether some data has been published; then, the events are pulled, and the function is invoked with the proper input. The responsibility, with this approach, is on the Lambda service, that must be allowed to read the stream, by setting an appropriate IAM Role.



For all other services, and actually for DynamoDB too, streams are not generated and so cannot be used to publish events. The event-source mapping is maintained within the service firing the event, that altogether holds the responsibility for invoking the function: it needs to be provided an appropriate IAM Role that is granted the permission to fire an Invoke event on the specific Lambda function.



3.2 Saga

It has already been discussed how to deal with cross-concerning transactions, that is, those changes in data spanning various contexts, that cannot provide a predefined set of data changes but that, instead, follow different paths based on the data stored in each context. Working with microservices, a smart solution that has been proposed is to apply, on design phase, the Saga pattern. It consists in engineering a flow of data actions that is either

orchestrated by a third component or choreographed and handled by the interested actors. In both cases, a Pub/Sub pattern for delivering messages is advised to be used.
Considering the AWS services that compose an AWS-hosted serverless architecture, there's plenty to choose from. Almost any event that can trigger a function, in fact, could be used to choreograph a saga.

Removing from the range of choices the uncomfortable ones, e.g. writing files on S3 and triggering events by doing this, these are the most common options:

- using SNS to publish events that functions are subscribed to and using the message as input
- exploiting DynamoDB events on the new data inserted
- using an orchestrator instead of a choreography and so making use of AWS Step Functions

In the case of triggering Lambda by using DynamoDB, it must be taken in consideration the possibility that many rows in low time are written, so it should be decided whether to set as the maximum rows to be collected before invoking the lambda function to one or to design the lambda function to possibly receive as input a set of data instead of data referring to a single entity. The configuration to be set following this approach is not so straight-forward. It must be taken in account that the chosen DynamoDB Table needs to be allowed to invoke the lambda function, thus binding two possibly cross-concerning entities. Furthermore, in some cases the event fired is not strictly logically bound to the storing or updating of some data. To by-pass both these two issues, it can be set up an *Events* DynamoDB Table. Making a table like this, though, can be more troublesome that useful: having a single huge table forces the lambda functions to filter out the events they are not interested in, causing a huge amount of useless invocations that consist in checking if the event is interesting and, if not, dying; so, it should be taken in consideration to create several events tables, each covering various events, possibly bound by the same context source or destination, up to a table per event. In this case, the overall efficiency increases (there should be less misses, down to 0 if a table per event is created), but the cost caused both by the cost of the service and the cost of maintenance can become unsustainable.

The best situation where to apply the triggering by DynamoDB streams or events is when the event is the writing of the data itself. In this way, the overhead in cost and in configuration is the least possible, even considering the other event sources that can be exploited to fire a lambda function invocation.

For the other cases, if a Choreograph-based coordination has been chosen to apply the saga pattern, exploiting SNS should be considered. SNS is a service coordinating Pub/Sub messages and channels, allowing the creation of specific topics. Alike the case where DynamoDB is used providing events tables, the numerosity of topics can be appropriately designed to adjust the ratio between configuration/maintenance cost and the cost relative to useless lambda functions invocations. It should be taken in account, though, that there is no cost for delivering messages to fire the invocation of functions, but only for the invocation itself, nor there is a service cost for the existence of a topic. The easy outcome from these considerations is that there is hardly a reason not to create a topic per event, even though the system gets bigger in the number of components and the feeling is that it might be a waste to only have a subscriber per topic.

All these choreograph-based coordination techniques for building a saga, have a noticeable problem: lacking a single point where the coordination happens, and consisting of many distributed components, removing a saga or changing it someway is not immediate.

Leaving behind choreography, the real alternative to DynamoDB events and SNS topic subscriptions is to have an orchestrator taking charge of the job of dispatching messages and getting under control the entire saga. For this purpose, there are various possibilities, but the winning solution is to create a Step Functions state machine per saga. Actually, in all the cases some sort of finite state machine is created and put in place.

A first approach to be considered is to have a lambda function, not belonging to any bounded context, whose job is to create and follow a flow by successively synchronously invoking the specific lambda functions performing the required job. A such created lambda function should:

- be allowed to invoke the functions
- contain the business logic that lets it decide the path to follow
- complete its job in the execution time limit

This last is the weak point of this solution: in AWS, lambda functions have an execution time limit that, by itself, is enough to complete state machines representing most use cases. The real problem is that this lambda function could possibly be paired with an API Gateway entry-point, whose execution time limit is very lower, that is, 30 seconds. This limit could be bypassed by invoking the orchestrator function asynchronously, thus moving up the maximum execution duration to 300 seconds. The other real issue is that a finite state machine must be represented, and all the cases must be treated, such as lambda functions failing. Finally, special cases like waiting some time or event before continuing the state machine is troublesome or impossible to achieve.
And here comes in help Step Functions:

- it is cheap
- the creation of a finite state machine is its business
- it has not almost any execution limit

Building a saga by using Step Functions consists in defining the state machine by using the Amazon States Language (ASL) and granting it all the permissions to let it invoke the lambda functions representing the tasks of the state machine. Each saga is then represented by a single Step Functions, having all the information inside and in a single place, thus helping the maintenance and the updating of it. Changing the structure of the state machine of a saga, for example by adding a state, only consists of updating the ASL code representing it. There is a minor problem that should be kept in mind: Step Functions can only be invoked asynchronously. This service offers some APIs to check the health state and the result of an execution of a Step Functions instance.
Providing synchronous execution is possible and can be obtained by having a Lambda function start the execution of SF, then polling the APIs to check the state, and then returning the result to the caller. This solution is de facto the creation of a lambda-based Orchestrator where the handling of the state machine is relegated to Step Functions, and so it must respect some of the previously mentioned limits of this solution.
Though the focus into this paragraph was on the coordination between transactions, it must be kept in mind that one of the strong point in this saga pattern is that when designing a multi-context transaction, they must be designed, along with the main flow's transactions, also some compensating transactions that undo the changes that were made by the preceding

local transactions. Anyway, this is more related to the design phase than on the technology implementing it.

3.3 Authentication

AWS offers a provisioned service, IAM, that allows to define fine-grained access to AWS services and resources, by defining roles, groups and users. By associating this service with Cognito, a world of possibilities is open for the designer to authenticate and authorize users and personalize their experience using the service. Cognito authentication can also be associated both with API Gateway to authorize requests to methods, and to AppSync, for authorizing and authenticating (and customizing) queries at any level of depth in the GraphQL structure.

In some cases, though, the designer might want not to exploit IAM and Cognito. Maybe an authorization service has been already developed and it is available or there is the need to develop an unorthodox kind of authorization. It is hard to find some cases where these AWS provisioned services do not meet the developers' use cases.

Anyway, AWS lets the developers not to use IAM in favour of custom authorizer. In some cases, it is quite easy and straight-forward, in others the integration is more cumbersome.

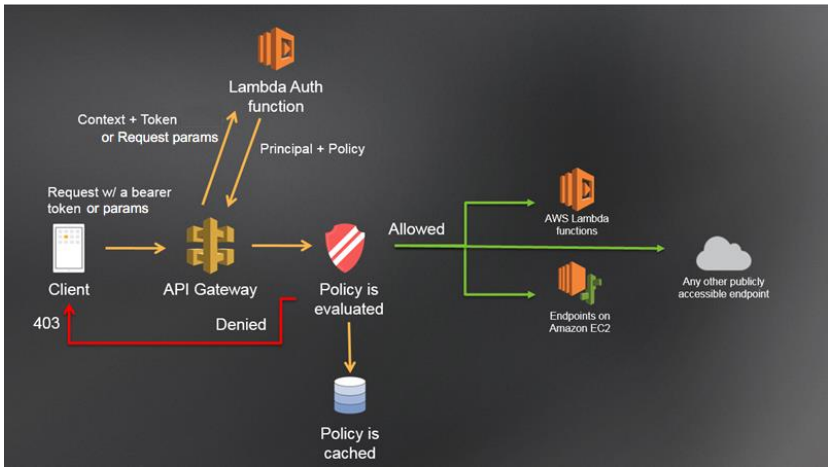


FIGURE 1 AWS SCHEMA FOR API GATEWAY AUTHENTICATION

When defining the authorization mechanisms to access HTTP Methods exposed by API Gateway, it can be set a Lambda function as authorizer. In this case, a specific data format must be followed for both the input and the output of the function. It can be set on the API Gateway to cache the result for some time, up to 5 minutes, to alleviate the load on the function.

The business logic of such an Authorizer is totally up to the designer that can choose whatever protocol or pattern he prefers. One of this that needs to be mentioned is jwt.

3.3.1 jwt

JSON Web Token is JSON-based open standard (RFC 7519) for creating access tokens that assert some claims. Clients presenting this jwt object will be trusted that the claims contained in the object are true, without the need to go finding the rights that are associated to the specific client making the request. This can work because the json object, the token, is signed by the releaser's private key, so if the releaser is trusted, then they are the token and the claims coming with it.

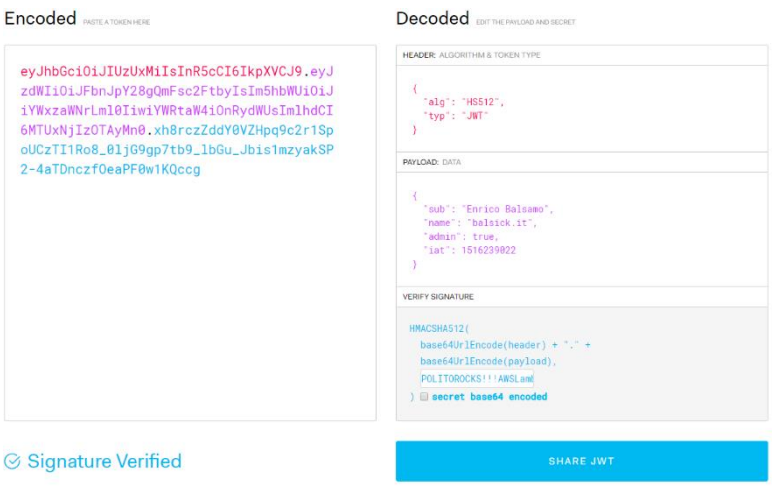


FIGURE 3 JWT.IO VERIFYING AND DECODING A VALID JWT TOKEN

The token is actually built of three parts separated by a dot: the header, a json object encrypted in Base64Url containing the informations about the encryption algorithm, the payload, a json object encrypted in Base64Url containing the claims, and the signature, that is obtained by applying the chosen digest algorithm on the preceding part, that is, the encoded header, a dot, and the payload, by using a secret private key. This private key can either be a simmetric key in case of simmetric algorithm like HMAC, or the private key in a public-key algorithm like RSA.

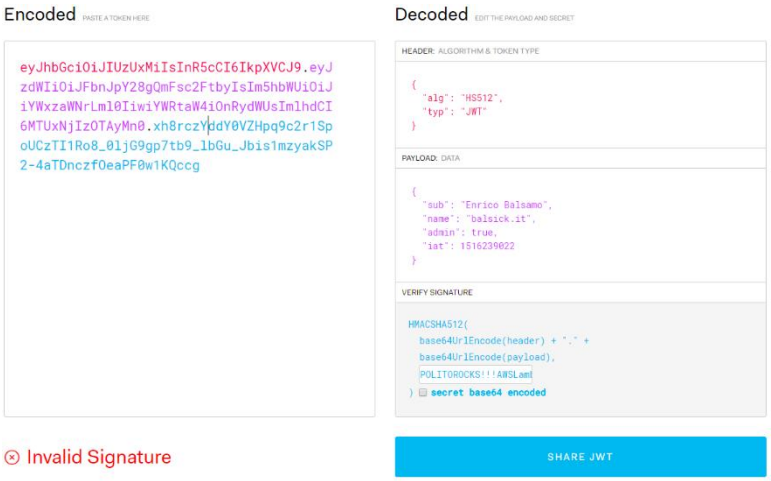


FIGURE 2 JWT.IO VERIFYING AND DECODING AN INVALID JWT TOKEN

When a jwt arrives along with a request, first the signature is verified by applying the proper algorithm. If it matches, the claims contained in the payload can be used right ahead to decide whether to provide the service or not. Claims cannot be changed without knowing the secret private key signing the signature, so the token can travel safely on a public network without being compromised by a malicious part. The token itself does not guarantee in any way that a

man in the middle steals the token: some more caution has to be used, for example by providing inside the token the ip of the client who obtained the token, or, more safely, by making the token travel on secure HTTPS channel that provides all the security not covered by this protocol.

4 Prototype

After studying in deep what microservices are and how a Microservices Architecture is organized and represent data and actions, and then deepening how microservices can be exploited by deploying the application by applying serverless, and in particular exploiting all the services that Amazon makes available through their Web Services to build a serverless application, I have studied a prototypical application to really elaborate an application using these technologies, so that I could design a system exploiting the patterns previously described, and clash against the issues they want to resolve, meeting the possibility to decide whether one way or another was the best choice for my application. In some cases, I have forced a path so that I could try out by myself a pattern or some services.
In this chapter, the prototype will be described and some of the problems faced will be dealt with closely. These will deal both with technologies and with design.

4.1 Used Book Store: the prototype

UBS is a platform on which users can find their used books they are not interested in anymore and find the used books they are looking for at advantage prices, without paying anything.
Books can in fact be purchased by paying with an internal virtual currency that flow within the application moving from an account to another one: on UBS you sell books for user book score (ubs) that you can use to purchase books from other users. The seller decides the price, that is, the amount of ubs he values his book, and the other users who want to purchase the book will compare all the offers, that is, the selling proposals, to choose who to buy from. If a user is short of ubs, he can purchase some for real money. Each transaction in ubs is taxed, and the tax is calculated based on the frequency of purchases and sales the interested users perform.
A user can purchase many books at a time, by collecting them in a cart before confirming and creating the order. Adding an item to the cart, though, doesn't guarantee the user that nobody else will be able to complete the purchase before him.

Couriers and deliveries are managed by UBS, in fact they collect the books to be shipped where the seller desires, and they deliver them to the purchasers. Couriers access the system through a dedicated portal by using their credentials. They get notified by email and notifications when the system assigns them a delivery. If they do not confirm or refuse the taking charge of it, all the couriers get notified that there is an unassigned delivery they can take charge of, by logging in and auto-assigning it to them.

4.2 Bounded Contexts

When discussing Bounded Context pattern, one of the key concepts that was expressed was the different meaning a word could take when changing context. This is valid on the contrary too: the same object can be defined and called differently on different contexts.

A remarkable example is the book: the UBS department dealing with the books catalogue can effectively define it book, meaning what an author wrote; in the department where the users' offers are dealt with, a book defines exactly the item sold by a user, instead; into the cart, it will only be an item, because the cart is not interested in what it is, it prefers to only know how many instances of the same item it contains and at most the price of each, and the same applies for who works with orders and invoices; couriers will probably define it goods before packing it, then it will match with the parcel if it is the only item.

Having these concepts clear in mind is strongly advised to be able to comprehend in depth this pattern, the reasons why it is important to apply it, and be free from any chain that could make the microservices be logically bond one-another that make the system design and development harder than the necessary.

With this being said, there is not a rule that makes the division of the system in contexts one way only.

I personally came out with these contexts:

- Book
    - Book-Review
- User
- Offer

- Cart
- Order
- Shipping

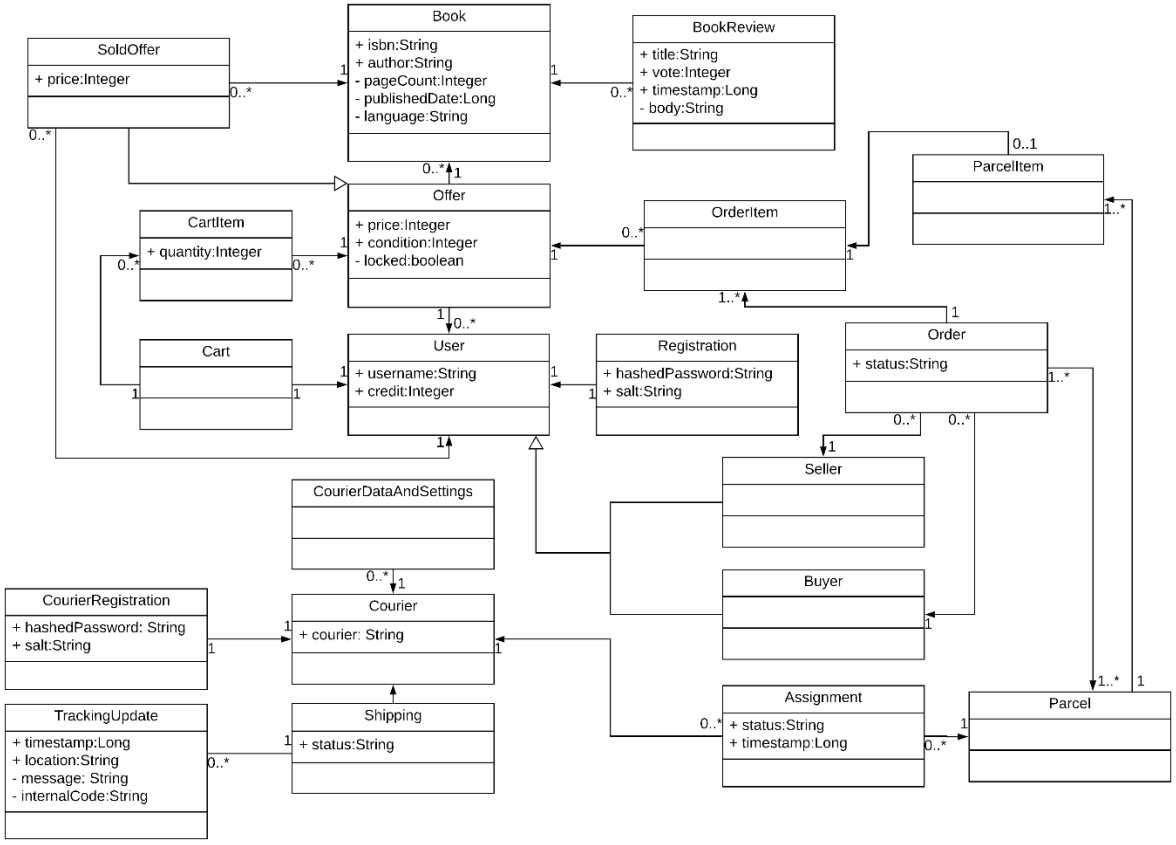


FIGURE 4 UML CLASS DIAGRAM REPRESENTING THE OVERALL CLASSES NEEDED FOR THE PURCHASE PROCESS

4.2.1 Book

This context deals with the catalogue. The key object is the book, defined as what an author, or more than one, wrote, in a specific edition. Practically, it is the logical item uniquely identified by an ISBN. And it is exactly the ISBN that in UBS uniquely identifies a *book* in the Book context.

UBS is not a www colossus, so at any moment in time it does not contain in its internal database all the information on all the editions of all the books ever written and put in commerce. It only stores data about the books that have ever been in sale on it by using Google Books publicly available APIs (more on this later).

Internally books are stored onto a DynamoDB Table having as primary key the isbn. An additional index is specified, built on the main author as partition key and isbn as range key.

4.2.1.1 Book-Review

Defined as a sub-context of Book, it deals with the reviews the users can write about books. This context shares with the super one the concept of book. With the books lifecycle as previously defined, it comes free (and natural) that a book review can exist only if it was ever put in sale in UBS.

Data is stored on a DynamoDB Table having as partition key the *isbn* of the book the review refers to, and an autogenerated **uuid** (universally unique identifier) as range key. This uuid could identify a specific review by itself, being universally unique, but an index with such a composition grants better performances on most of the queries that can be performed on the table.

4.2.2 User

This context deals with the management of users, both sellers and buyers. In UBS in fact it does not exist any separation between these two categories: anybody can both sell and buy books from other users. A user owns some ubs, that is, a credit he can use to purchase books and he can fill up by selling books or by buying with real world money. The DynamoDB Table representing a user object is as so defined by only its username**,** uniquely identifying the user within the system, and the credit associated to it. Additionally, a reputation score is stored as attribute: this information is refreshed and recalculated to reflect how good of a user it is. It increases by correctly using the application and decreases for misbehaviours, and it is used to properly calculate the tax collected on the transactions involving him.

In this context also information about authentication reside, on a separated Table called *Registrations*. Credentials are stored as a set of 3 attributes:

- username
- hashedPassword, that is the result of applying a hashing algorithm to the user's password and the
- salt, a random-length (between 8 and 16) randomly generated alphanumeric string

When the user logs in, he communicates his username and his password: the system applies the same hashing algorithm to the password by using the salt associated to his username, and whether allows or denies access.

4.2.3 Offer

In UBS, an offer is a specific entity put in sale by a user. Here *'book'* changes meaning, and it refers to the specific item that is put on sale. An offer may be considered a composed entity made of the seller identifier, the book (meaning what Book means) identifier and a price. Additional characteristic can indeed be associated with this entity to increase the information about the book-item, such as its conditions and some pictures. Whether putting into it also some information on the location the item will be sold from or not is something to be thought. Based on the quantity of the same book a user can sell at a time, an **uuid** could be decided, in design phase, whether it is needed to uniquely identify the offer or not. Furthermore, keeping a unique identifier for this entity helps in some way the cross-concerning handling of this entity: as it will be later expressed, a reference to an offer is useful to be carried around the system.

In the DynamoDB Table this context's data is stored, I have decided not to use this just presented uuid as main partition key: the couple user-uuid compose the primary key for the table. Partition key and range key suggest DynamoDB how to store data in the storage, and this choice grants a better efficiency. Querying by uuid by itself is granted by an additional global index made only of it as hash key.

The table so far defined does not provide any information on the status of an offer, meaning whether the offer is still on sale, if it has been cancelled or finally sold. Adding an attribute to the DynamoDB Table's Item is not a viable solution: using a relational database adding a column could be considered and put in place without adding consistent inefficiencies, on the contrary it would have been simplified the overall system's simplicity. On DynamoDB there is not a way to query only data having a specific value on an attribute: the most efficient way to achieve this is to tell the SDK to filter out the results not satisfying some predicate, but all the items need to be retrieved first, it is a client-side operation. This means consuming throughput without gaining any advantage by such a data architecture. On the other side, if this *Offers* table was small, it should have been considered that maybe the most efficient way was to add an attribute and use it to perform queries.

For these reasons, I have decided to define an additional DynamoDB Table, *SoldOffers*, where *Offers*' item is moved to after the sale has been processed. This data definition totally matches *Offers*' one.

4.2.4 Cart

A cart is a bond between some items and a user before he completes the purchase. By applying this definition, the context is defined both in data and in actions that can be executed on it.
Particularly for this context's data, it does not exist a unique way of storing. The technology chosen for this purpose, in this case DynamoDB, suggests a direction to be followed but does nothing more. If a technology that declaratively better perform with one data-structure or another was chosen, the range of choice would close and the designer would be forced.
Going straight to the point, at least these choices to handle a cart system show themselves:

1. an item representing a cart in its whole
2. an item per item contained into a cart

By using a relational database, only the retrieving of the cart is efficient with the first option, but there is a great disadvantage in all the other queries. In this case, the technology would force the choice.
DynamoDB on the other side has advantage with both the models. By storing an item structured as this

- username
- set of offers' identifiers

retrieving of the entire cart is immediate, adding or removing an item from a cart is efficient, checking if a user's cart contains an item is simple by using the *CONTAINS* operator in the condition on the SDK, but it is a client-side operation. And this is the reason why I've not chosen this model. In some occasions, in fact, there might be the need to remove an item from all the current carts contents. Performing this operation requires a full table scan to retrieve all the items, then filtered by the SDK to get only the items containing the indicated item, the removing in memory of the item from the cart, and the updating of the just changed items on the Table.
By using the second data model, thus putting an item per each user-offer couple, the execution of such an action is hugely easier: it consists of retrieving the interested items and deleting them. If DynamoDB allowed to perform deletion on global secondary indexes or on incomplete primary keys, this action would come almost free. Adding or removing a specific item from a defined user's cart are immediate actions. The only downside with respect to the

first data model is the retrieving of the cart, that requires a reconstruction of a single object from many items. This is acceptable, though.

A Carts Table's item, in UBS, consists of a primary key made of username and Offer's uuid, and the quantity of such an item contained in the cart. A global secondary index made of the offer's uuid as hash key is defined too, to allow queries where username is not available.

4.2.5 Order

After the user explored the available books, the offers for each book, and maybe by the sellers' ratings chose some offer and added them to the cart, eventually the cart becomes an order. So, this context deals with everything that happens between the moment the user decides to make the purchase and when he obtains the items he desired.

An order is a bond between the buyer user, the seller user, and the items that the transaction is involving, or it has involved. This logical disjunction suggests the presence of something that indicates the status of the order. Unlike the Offer, where an additional table stores the offers not in commerce anymore, for this context's main data model I have considered sufficient an attribute representing the status of the order. There is not a catalogue of orders that needs to be browsed only if they are in a determined status: having all the orders in a unique list regardless of their status is what users expect.

Exactly like Cart, it must be created a bond with a set of items: unlike Cart, though, the items associated with an order are immutable. The need to perform queries to alter this set drops, and so they can be an attribute of a single item that is enough to handle an entire order by itself.

Summing up, orders are stored into a DynamoDB Table having an automatically generated **uuid** as primary key. Two additional global secondary indexes are defined, each having one of the two interested users in the transaction as hash key and the status as range key. Furthermore, the total cost and the assigned tracking, if any, are stored.

4.2.6 Shipment

This context deals with all the information managed by the hidden type of users of UBS, that is, the couriers. In fact, they use the application to declare the assignment of a shipment, the picking up of a parcel, the final delivery. To make this possible, a separate authentication

system is present in UBS and managed into this context. The authentication process matches the one in User Context.

The data model for this context, though, is quite more complex than those so far described. Let's analyse the data flow (it will be the main argument of a following paragraph):

1. an Order gets payed for and is ready to be picked up
2. the system chooses what courier to assign the parcel to and notifies it
3. the assignment can be either confirmed or refused, or no action is taken by the courier in some time; anyway, a courier picks the parcel up
4. the parcel travels and a tracking system shows step by step the actions on the parcel
5. the parcel arrives at destination

To satisfy such a flow, at least three entities must be defined:

- Assignments
- Shippings
- Trackings

By using these three entities, all the delivery process should be covered.

Assignments deals with the first phase of the process, prior to the final picking up by a courier of the parcel that must be delivered. Then, a shipping entity is created and whenever the courier needs to publish some event about the parcel's travel, an update is pushed into Trackings.

So DynamoDB will handle for this context a Table for each of these entities defined so far.

Assignments table's items are identified by using a primary key made of the courier as partition key and an autogenerated *assignment* uuid that is actually uniquely identifying the item without the need to exploit *courier*. A primary key defined as such, anyway, helps with queries. The *parcel*'s id, a status and the timestamp of the moment of the assignments are stored along. The couple made of the parcel identifier and the courier on most cases only exist on one item at a time. A courier should not have two assignments for the same parcel. On some occasions, though, there might be the need to ignore this limit: this is the reason why an assignment uuid needs to exist. A global secondary index made of status as hash key and courier as range key is useful to be defined, because it allows some queries to be more efficient.

Shippings' items are defined by a primary key made of courier as partition key and an autogenerated uuid, *shipping*, as range key. Reusing Assignments' *assignment* autogenerated key would have been possible, but it is logically incorrect because there exists no bond between the two attributes. The identifier of the parcel and a status are stored into the item too.
Trackings' items consist of a primary key totally matching Shippings' one, and a set of updates. An update is an object made of a timestamp, an optional code identifying a type of update that the courier handles internally into its system, the location and a message.

When being notified about a pending shipment, couriers need to know some information on the parcel they are assigned to delivery. This information consists at least of physical information, that is, the three measures and the weight, the address they must pick up it from and the address they must deliver it to. Because of the assignment process, putting this information into Assignments or into Shippings is wrong and unfeasible. For this purpose, a ParcelInfo DynamoDB Table is defined to contain all this information.

Still for the assignment process, the system needs to persist some information that allow some rules to be applied to decide what courier to assign the parcel's delivery to at first. In this case, I have decided not to build a spaceship, doing something very basic: a circular buffer moving its index on each assignment. To make it work, the circular buffer needs to be stored, alongside with the last used index. Some more data and settings could need to be saved, so I have defined a *CouriersDataAndSettings* DynamoDB Table having a designer defined *dataKey* as partition key and a *data* object as data.
Into this Context also the demographic information about the couriers resides, consisting of a unique id being the courier name, a visual name, an email where to send notifications, an email and a phone number where an UBS system admin can contact them, and a score representing the courier's reliability. This data is stored into an appropriate DynamoDB Table, *Couriers*.

Finally, a *CourierRegistrations* DynamoDB Table stores the information needed to handle the registration and login processes. This table's items consist of the courier identifier as partition key, a hashed passphrase and the salt used to hash the original passphrase.

4.3 Distributed transactions and saga

In UBS, like in every application divided in contexts, the execution of distributed transactions is an issue and it must be treated on time. *On time* means in the design phase through the integration phase. When designing a context and defining its boundaries, the APIs exposed, that is, the doors left open for using the data, must be accounted for. This does not mean that the data model should reflect the APIs exposed, but that it should accommodate them. When integrating several contexts altogether APIs might not be sufficient to satisfy a dataflow. In fact, as already expressed several times in the preceding chapters, contexts should not know each other. A careful reader might have noticed that in the paragraph about Shipping Bounded Context, there is hardly a reference to the keywords used when describing the preceding contexts; a new word actually arises, *parcel*, to identify what is probably the order in Order Context. The transformation of the order in parcel is one of the flows that will be described in this paragraph.

As already seen in the previous chapter, AWS makes available various ways to lead a saga, proposing SNS to dispatch messages, DynamoDB Table's triggers to fire events when data is stored, and Step Functions to orchestrate a distributed transaction.

4.3.1 The product availability problem

Some of the problems related to the management of a warehouse exist in UBS too, despite the word *warehouse* has not been used in describing UBS, and the reason for this being the lack of an actual warehouse where items on sale are stored.

On UBS items being on sale are real objects. This means that they are finite, and their availability is indeed low too. So much low that for each item there is only one instance put on sale. But before considering the specific, boundary, case, let's discuss a standard warehouse problem.

In a system including a Warehouse Context managing the data of a warehouse, the data model would be something like this:

- item identifier
- quantity
- position

Let's ignore the position and focus on the couple id-quantity. An item's stock can be refilled when the provider sends some to be sold, and consumed sale after sale. Starting at 1000, after a sale, the quantity becomes 999. So far so good. Sale after sale, though, the stock becomes lower and lower. If the provider does not refill the warehouse, some problems might occur: what happens when two people try to purchase the last item at approximately the same time? It depends on how the purchase flow is managed.

Going back to UBS, the problem clearly remains: the total amount of the same item that can be stocked at some time is not useful data; the problem is when the punctual amount is singular.

Considering the solutions presented when discussing about distributed transactions and saga, various solutions show up and they are built around Compensating Transactions pattern.

In a platform where the purchase is not mandatory to be performed right-away, but instead the item chosen can be added to a cart without any warranty or obligation, this problem is

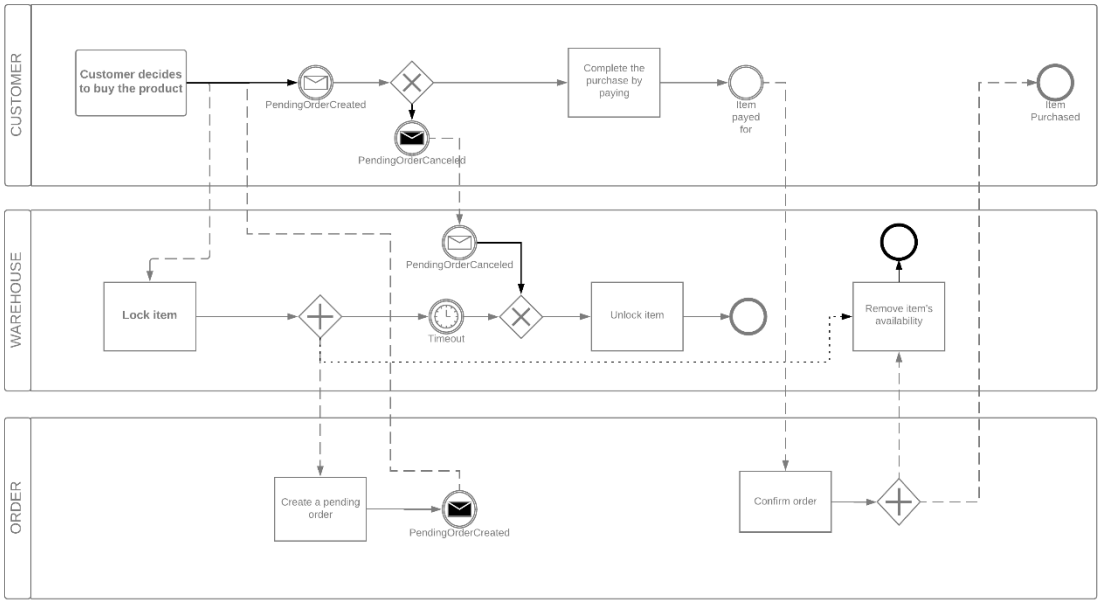


FIGURE 5 BPMN REPRESENTING THE PURCHASE FLOW IN A SYSTEM HAVING NO CART

harder. Let's consider, in fact, the opposite case, that is, there is no cart. In this case the user chooses an item and starts the transaction to purchase it. He is giving his word to the system that he will most probably complete the transaction. The system, in this case, can be designed so that when this transaction starts, the item is locked and nobody else can make the purchase. A timeout grants the unlock of the item if the user does not respect his word.

If the system does not include presale as a possibility, then the workflow becomes more cumbersome. In fact, checking if an item can be purchased is no more feasible first. The

process now involves an additional object, the cart, where at some time some no-more-purchasable items could be present. To keep the service consistent, when an item's stock becomes 0 it should be removed from the carts it resides into.

So now the transaction involves 4 different contexts, and the flow bounces from one to another. Using an orchestrator and defining a finite state machine makes it easier than sticking to events.

In this case, the orchestrator is AWS Step Functions, and the visual finite state machine is the one provided by the service after configuring it via ASL.

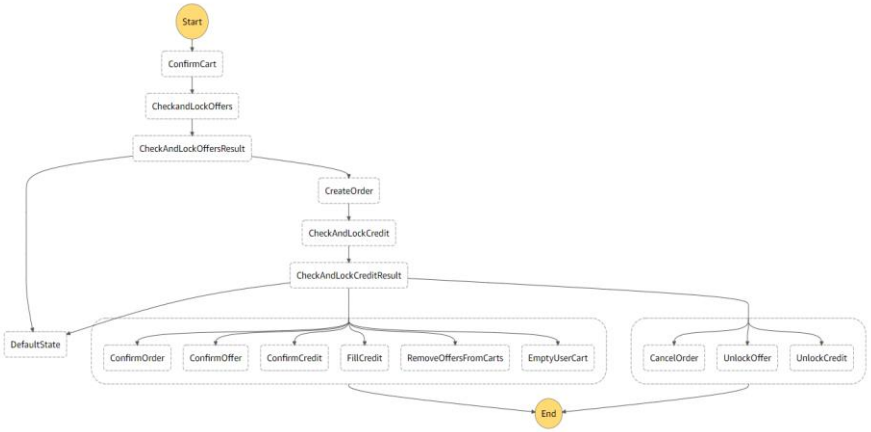


FIGURE 6 AWS CONSOLE SHOWING A GRAPHICAL VIEW OF AN AWS STEP FUNCTIONS WRITTEN IN ASL REPRESENTING UBS'S PURCHASE FLOW

A first iteration of the flow is the one depicted in the image above. In this state machine the service lock first the offers, then the user's ubs (credit) and, in case of success, completes the execution by confirming the previously put locks. If whether the locking of the offers fail because one of them is already interested in some transactions, or if the locking of the credit fails because it is not sufficient to complete the purchase, the modifications so far performed are cancelled and the data is restored.

Though working, the described model does not respect the saga pattern using the compensating transactions pattern. In fact, it states that first the modification should be performed, then, in case of failure, reverted: the system should be eventually consistent, the consistency during the transaction is not important.

Fully applying this pattern to this process would lead to a finite state machine like the one depicted below, where the readability is really improved and the quantity of data flowing from and to persistency objects is reduced.

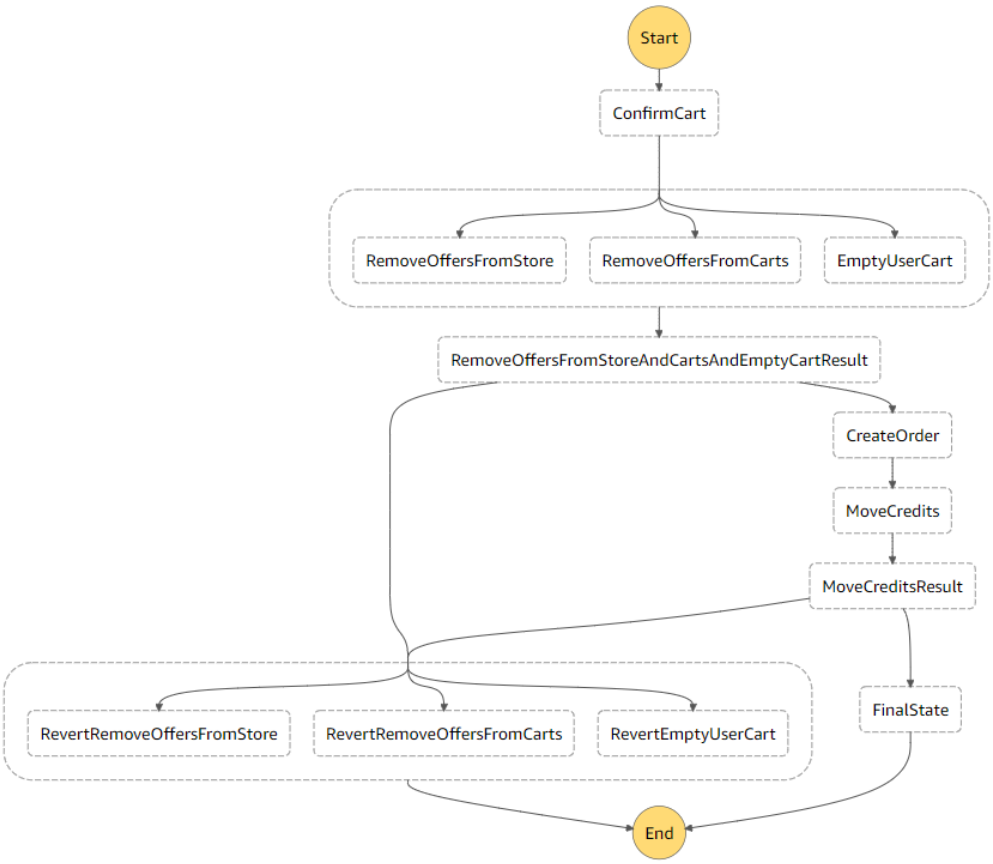


FIGURE 7 AWS CONSOLE SHOWING A GRAPHICAL VIEW OF AN AWS STEP FUNCTIONS WRITTEN IN ASL REPRESENTING AN HYPOTHETICAL UBS'S PURCHASE FLOW FULLY COMPLIANT TO COMPENSATING TRANSACTION PATTERN

First the cart is confirmed: this is only a renaming of a getting cart operation. No changes are applied during this task. Then the offers are removed from the store, from all the other carts containing them and the user's cart is emptied. These tasks are executed independently in parallel. Each one of them does not mind the possibility that the other tasks might fail. It is an optimistic pattern indeed. Depending on the overall result of these tasks, either the process goes on or starts compensating transactions, one for each just completed ones.

Some attention should be paid on the fact that the compensating transaction should actually do nothing if the transaction its compensation work on failed. If there was no failure, the finite state machine moves to a state representing a task moving credit from the purchaser to the seller. If it fails, the same previously described compensating transactions will take place and

restore the system to the previous consistent state. Otherwise, the system reaches a new consistent state.

MoveCredits task can be internally represented by a transaction performing two separate actions:

1. update the purchaser's credit by decreasing it by the amount to be paid
2. update the seller's credit by increasing it by the same amount, minus the tax

Increasing the seller's credit has no points of failure, there is not any constraint to be satisfied to keep data consistent. On the opposite, decreasing the purchaser's credit can only be performed if it is greater than the amount to be decreased, otherwise it will fail. In this case DynamoDB API comes to help, because they allow conditional updates to be performed, helping in reducing concurrent modifications. In the very unlikely event that two different purchases are performed by the same user and that the credit only allows for one purchase to be performed, by using conditional updates one of the two purchases will fail. Furthermore, by failing in this atomic way there is no need, in this specific case, to apply any compensating transaction because the data has not changed.

4.3.2 Shipment transactions

In UBS users who want to sell or purchase books do not need to worry about making arrangements with the counterpart, nor going to a post office when they need to send the books they sold: managing shipments is up to UBS itself.

Because of this, UBS needs to automatically assign jobs to couriers, providing them the required information for them to arrange the delivery. As mentioned in the Shipment Bounded Context, UBS's algorithm for doing this is very simple because it is based on a circular buffer. In that same paragraph, it was underlined how the workflow representing the shipment process needs various entities to be represented and persisted. But it actually interests many contexts too, making most of the transactions regarding Shipment cross-concern, and this makes it interesting to be studied in deep.

First, the assignment of the parcel starts in the Order context: it is in fact the completion of the purchase, thus of the order, that defines one or more parcel.

UBS is a distributed bookstore not storing books into a warehouse but at the sellers' houses. This means that when a user purchases some books, they most probably will be delivered from different places: an order will consist of many shipments, and so of many parcels.

After the payment, the order is confirmed, and many parcels defined. For each of them, an event gets published, leaving Order context. For this case, I have preferred to exploit AWS SNS rather than coordinating the transaction with an orchestrator: there is not any saga to apply, it is the perfect scenario where to apply Pub/Sub pattern.

Then, the focus moves to the Shipment context: the parcel needs to be assigned to a courier. It has been already stated that the assignation of a parcel to a courier is not a definitive action, as the courier can refuse or be late in accepting it. Because of this, a flow is defined:

1. The system automatically generates the first assignation
2. If the courier refuses it, or if a timeout expires
    a. All the other couriers get notified of a pending assignment
    b. They are given the possibility to assign it to themselves
3. The parcel is assigned to a courier

Transposed as BPMN, the need of an orchestration results glaring.

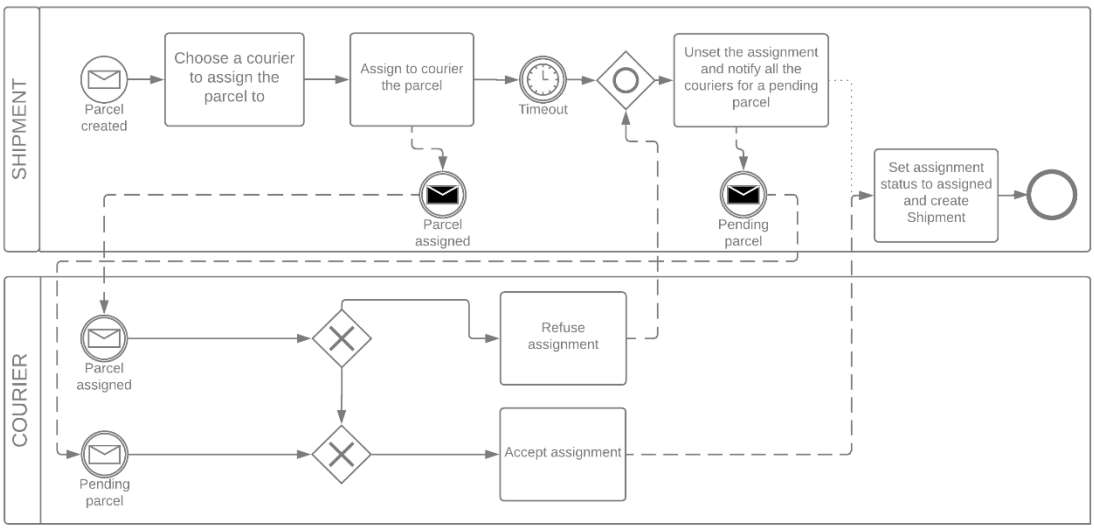


FIGURE 8 BPMN REPRESENTING UBS' SHIPPING ASSIGNMENT PROCESS

Nevertheless, the orchestration only handles part of this, yet partial, flow. In fact, the interaction with the courier is asynchronous: an AWS Step Functions state machine should not wait, and cannot normally wait, for the user to interact with the system. The only state

interested in doing this is the Timer task handling the timeout, but it is a necessary exception and it does not *directly* do that.

The orchestrated part of the total flow helps solving the problem of the timeout. The other solutions were to either not including a timeout, thus having the system only let a courier accept or refuse, or not pre-assigning a courier to the parcel. With the first solution, if a courier ignored the notification telling them that a pending assignment was waiting for their acceptance, neither a human-acceptable latency nor the certainty that the parcel would be ever dispatched are granted. The second solution is acceptable but can cause starvation in some couriers that might be not fast enough to accept any job.

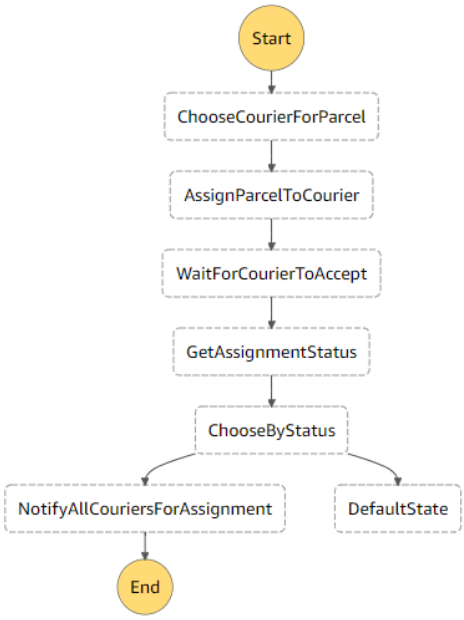


FIGURE 9 AWS CONSOLE SHOWING A GRAPHICAL VIEW OF AN AWS STEP FUNCTIONS WRITTEN IN ASL REPRESENTING UBS'S SHIPPING ASSIGNMENT PROCESS

Unlike the purchase flow where a finite state machine is defined to exploit a saga distributed transaction, this state machine does not contain any distributed transaction at all: it is only a different usage for the same technologies.

After the shipment is assigned, the actual delivery needs to be performed. The Shipping Context's API allow the courier to declare some updates that compose the tracking system. A user might desire to be notified for each of these updates. For this to be possible some scenarios can be considered:

- When inserting an update into DynamoDB, it is contextually pushed to SNS for client notifications and to SES for emails
- The generation of the notifications is separated from the insertion of the data

The first is very simple to accomplish and quite straightforward.

The second is useless and carries no real advantage with it, but still it is interesting to be watched closer. In fact, to do it three different AWS components need to be created. The flow should be this:

1. The courier causes an insertion into DynamoDB through some APIs
2. DynamoDB triggers by event sourcing the invocation of a Lambda Function
3. The Lambda Function dispatches notifications to the interested user(s) by using SNS and SES

The only real usage for this approach is in the case when the APIs storing into DynamoDB the updates cannot be updated or integrated with SNS and SES. In this unlikely case, the strength of the Microservices architecture intervene and some new characteristic can be appended to the system.

In the end, the courier deliveries the parcel, and the shipment is completed. Apart from generating the required notifications, and this is equal to the preceding example, the order that the parcel came from needs to be updated because also it is now completed.
The usage of Step Functions to put in place a finite state machine is out of the table here too, the time interval such a transaction spans is too high. So, both because it's nice to close the circle as it started and because it is the smartest thing to do, an SNS event published by Shipment can be used to notify to Order the completion of the order.

4.4 Considerations on API

In the preceding paragraph, and particularly in the example on the separation of responsibility between the component updating the data and the one generating notifications, I intendedly omitted any reference to the interface interposed between the user and the data transformation. And by this I don't mean the UI the user interacted with, but the layer just below that.
Despite the fact that, starting from the 1st chapter, the API Gateway has been indicated like the component having the responsibility of putting in communication the external world with the business core, and that an **AWS API Gateway** service exists keeping the same name, it is not the only choice available. GraphQL exists and **AWS AppSync** service has been created to implement it, and it is an exceptionally good alternative.
Between the two alternatives, there is not a precise winner, but some distinctions to help in deciding whether to use one or another can be made: as always, the result tells that the choice is situational.

RESTful APIs have now been on top of the discussion table for several years, and they have become the de facto standard for designing APIs. Integrating services among them is becoming, maybe even thanks to the microservices approach spreading, a daily matter, even in the industry for making machines communicate with MES and ERP software.

4.4.1 AWS AppSync: the anti-REST

GraphQL is the only real alternative for building as structured and human readable APIs as with the REST pattern but offers some advantage that make it preferable in some occasions. In fact, it allows the client to choose, at the request moment, what data it is interested to, and only of it, if available, the response will be made. This clearly makes the decision of which approach to follow pending to GraphQL in case when the client should reduce as most as possible the bandwidth usage: mobile phones are the main use-case for this technology.

AppSync provides a GraphQL end-point easily configurable specially to communicate with AWS services by using the default templates. Matching data with DynamoDB table is a matter of minutes.

Let's consider the previously discussed case when some notifications have to be fired when the courier insert some updates on a shipment. It had been proposed a way to separate the responsible for storing the updates into the persistency objects, DynamoDB, from the component firing the notifications, and it was said that in most cases this approach was not necessary. This was said having in mind an API Gateway based situation: already having a Lambda function being invoked for storing the data, there is little sense in creating and invoking one more for pushing notifications.

If the APIs were provided by a GraphQL application instead, following this approach would come in handy and be perfectly legit and the best. In fact, customizing the GraphQL mutation to also fire notifications would be easy if Apollo or an equivalent GraphQL gateway was used (it would only consist on changing the code run to handle the mutation), but on AppSync it is cumbersome and maybe almost impossible: the source for a query or a mutation is not writable. AppSync in fact only allows for linking a query to an AWS Service. The only way to do this would be by writing a Lambda function, thus losing most of the advantages of AppSync.

So, the choice of the technology providing the APIs leads to some choices in the design phase. In this case, the smartest solution is to attach to the DynamoDB table the shipment's tracking

updates are written a Lambda function invoked on each write operation. This Lambda function only handles the generation and the pushing of the required notifications.

In terms of cost, in this example the overall cost would be approximately the same of the opposite approach consisting in API Gateway invoking a Lambda that deals with the updating of the DynamoDB Table and the generation of the notifications, because in both cases there is the invocation of a single Lambda function and the writing into the storage, while both AppSync and API Gateway are free. The solutions staying in the middle between these two are more expensive because they involve the invocation of multiple Lambda functions instead of one.

4.4.2 AWS API Gateway: the classic choice

API Gateway is a pattern that is surely not born with microservices. In a monolith, for example on a Java Spring application running on a Tomcat, the methods mapped to various HTTP entry-points compose an API Gateway themselves, dispatching the outer requests to internal function calls.

On AWS, using API Gateway service allows for defining fine-grained APIs by setting resource by resource, method by method, either the Lambda function or the external HTTP destination or an internal service to be called.

Linking a method to a Lambda function is the standard choice but linking it to an AWS service is very interesting and allows for various scenarios.

In fact, some GraphQL-like behaviours can be obtained by exploiting this possibility: it can be mapped an HTTP call to check the authentication of the caller, then verify the data coming along with the call, and finally write the data passed, possibly transformed, directly to DynamoDB. All of this without even defining a Lambda Function. For most of the methods, this approach can be followed, greatly reducing the quantity of Lambda functions composing the system: most of what APIs do can be reduced to the CRUD actions, without any logic behind apart from checking authorization and data consistency. And in cases where this is not enough for handling the business logic, integration with a Lambda function is simple to achieve. When there is the need to start an orchestration as the executor of an incoming request, an integration with a Step Functions' state machine can be exploited.

4.4.3 The AppSync auth problem

When defining the entry-points to the application, they can be categorized: if applying a REST approach, first the resources need to be identified, then the methods for that resource, and finally it must be defined whether each method is public, private or requires authentication. In a standard GraphQL usage the authentication should be demanded to the web-server hosting it. It is a bad practice to demand the authentication to GraphQL because it must be managed by each query or mutation.

On AppSync it is about the same. The provisioned possibility consists on authenticating each query, by using an annotation-like system, that makes the query be accessible only if the request come from a client that is currently authenticated via IAM and matches specific IAM prerequisites, e.g. is into a Users' Pool.

If Cognito is not viable, or a custom authentication is desired, the configuration becomes trickier. AWS documentation presents this possibility but does not explore it in depth, but still a way can be found.

I explored this approach by using a Lambda function helping in the authorization phase.

Each entity in AppSync can be paired with an AWS Service set as resolver, spacing between Lambda Function, DynamoDB or ElasticSearch. Having the possibility to set a resolver per entity, when composing a structured entity many resolvers are called, both on cascade and in parallel, based on the entity's structure and on the usage of *query* or *mutation*.

How the auth can be achieved by using Lambda is not a one-way only, in fact there are two possibilities:

A. Using Lambda Function to authorize, authenticate and do the job
B. Let Lambda Function only deal with the auth phase

Both approaches have their advantages, moving the complexity from one-side to another. A lambda that handles the entire process is obviously more complex and the structure of the *query* of GraphQL calling results flatter and easier to comprehend. On the contrary, having a lambda function to only deal with authorization and authentication moves the complexity to the GraphQL schema.

Let's take as an example the Book-Review Context. In UBS its API are managed by an AppSync instance. Here, only reading reviews is publicly accessible, while the other three operations of the CRUD have restricted access.

A *BookReview* object is defined, containing all the required information.

Following the first approach, the query called to create a review has as resolver a Lambda Function that:

1. Checks if the request comes with an authorization token
2. If it does, checks the validity of the token, otherwise returns forbidden status
3. Extrapolates the user associated with the token, to authenticate the data
4. Save data to DynamoDB

The query only passes through the request, injecting the HTTP Header containing the authorization token, and translates the function's answer to a BookReview object. The configuration is quite straight-forward, but the Lambda function is bigger, both because it has to deal with the saving in the data to DynamoDB accordingly to the request, and because possibly it is created a single Lambda function for resolving many different AppSync queries: the alternative would be to create a Lambda function per query, but there would be too much of a repetition of the same code handling the authentication, that would lead to unsustainable maintenance too; in fact, if the authorization mechanism changed, each and every lambda function should be updated to meet the new requirements; furthermore, having a single Lambda function managing many similar requests helps with the system's readability.

When creating another authenticated entry-point applying the same approach, all that changes is the 4th point of the function's procedure. The second approach is based on this idea, and so the 4th point is moved out of what the functions does. The configuration of the query becomes harder: aside of *BookReview*, an additional *AuthCreateBookReview* object should be defined, made of the same data that BookReview is made of, plus an additional BookReview. To this inner *BookReview* it is associated a DynamoDB resolver that deals with the data insertion.

The query is configured to return an AuthCreateBookReview instance, and to this entity the Lambda function is associated as resolver. In the mapping template, it is injected as before the HTTP Header containing the authentication token.

How does it work? GraphQL calls the resolvers on cascade from the most outer to the most inner. When a resolver throws an Exception, the cascade stops. In this case, the resolver that fires the Lambda function, when translating the function's answer back to GraphQL, can throw

an *unauthenticated* Exception, avoiding the calling of the inner resolver inserting the review to DynamoDB.

4.4.4 Public Cart API – Part 1

One of the strong points in the microservices architecture and the global movement around it, is the fact that a service's APIs, after becoming stable and working, sometimes become public and reusable by other developers and services.

Having this in mind, I have made some arrangements to UBS Cart API to make it a service publicly available and reusable by third parties. Supporting this choice, various reasons from both the context itself and the microservices patterns show up:

- Cart is a Context totally independent from the others
- Cart only knows that there is a binding between an entity (the user) and various other entities (the content), each with a different multiplicity
- Cart's data does not depend on events but exclusively on API calls

This last point is very important because otherwise the coupling would not be enough loose to easily guarantee the possibility to make a service out of this context.

For this purpose, some changes to the data model should be applied: data is still stored into a single DynamoDB Table, but an additional attribute indicates the context the cart is valid into. Defining the 3rd party user as *domain,* and keeping the rest of the model, what differ are the primary key, the indexes and the access to the data itself. Using the user identifier as partition key is not anymore viable because it is not unique across the various *domains*.

Because of how a cart is represented composing many items in the table, data is now partitioned by using a composite key uniquely identifying a cart: it is made of a composition of the domain and the identifier of the user inside that domain.

Product remains a range key as it was before, ranging into a domain representing a single user's cart.

Cart requires in its API a method to act on all carts containing an item, keeping in mind that product is not unique among the entire service but only inside a *domain* too: this means that an index needs to be created, having as hash key a composite key made of the product and the *domain*; additionally, the user's identifier can be set as range key.

To make the APIs more complete, it should manage not only one copy only products but stock keeping unit (SKU) too. This forces the APIs to manage customization, storing and providing methods for the settings of each domain. Furthermore, even though a SKU with a provision of one can be considered a single copy product, the APIs provide fine-grained settings for each product.

Finally, the authorization phase changes too. It is no more the customer who executes query but an external service. This 3rd party is responsible for authorizing the user both internally and on UBS Cart: the service authorizes, by using its private credentials, one of its users. The jwt token Cart provides at login contains the domain as a claim and the user as the subject.

4.5 Considerations on CloudFormation

CloudFormation is a very powerful tool that can automate the process of the creation of a group of AWS Services (resources), a stack, providing the tools to create templates that can be customized when deployed.

Using it badly is easy, and I too have stepped against a wrong usage. In fact, at first, I have tried to define the entire platform supporting UBS into a single configuration template. This is wrong for various reasons:

- The readability of a file long more than 3000 lines is very low
- Such a single file leads to a monolith-oriented thinking, but UBS runs on a microservices architecture
- If CloudFormation is going to be the main delivery tool, creating multiple stacks helps with the continuous delivery because only the interested services are updated
- CloudFormation comes with a graphic tool both representing and usable for configuring the template; making a huge template makes the resulting graph totally incomprehensible (cfr. image below)

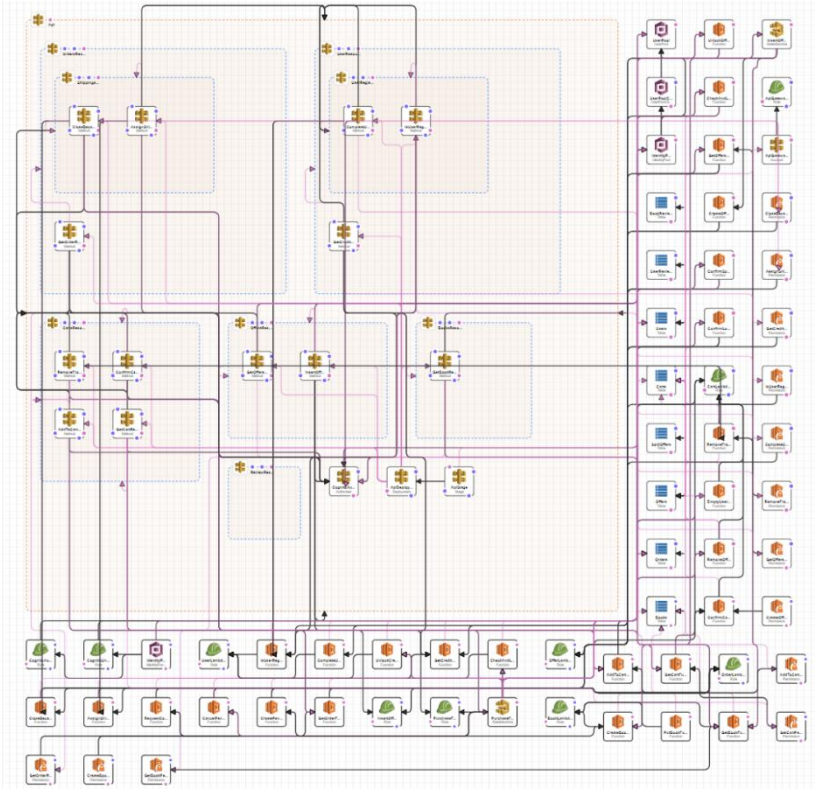


FIGURE 10 AWS CONSOLE SHOWING A GRAPHICAL VIEW OF AN A CLOUDFORMATION TEMPLATE REPRESENTING A CONFUSING AND INCOMPREHENSIBLE, YET PARTIAL, GRAPH OF AWS SERVICES AND THEIR DEPENDENCY TO DEPLOY UBS'S ARCHITECTURE

A more proper usage of this tool consists on applying the design patterns of microservices: it should be applied a division both on Bounded Contexts and on layer, so a stack should be created per layer per Bounded Context.

4.5.1 Public Cart API – Part 2

In the previous part of this paragraph, (cf. Public Cart API – Part 1), it has been proposed a way to make UBS's Cart API public and usable by 3rd party services. All the service's data lived together into a single DynamoDB Table and avoiding conflicts is based on the usage of some composite keys built on the domain name, the unique service's identifier.

It is a nice approach and it works, but when the services using these API become many, it can become slower and a little confusing. Furthermore, if these APIs have a cost for the service, with this approach it is cumbersome to identify the usage cost, unless registering every single API call into a registry.

An interesting alternative is provided by CloudFormation, because it allows the automation of the creation of stacks.

For these APIs, a stack built on the resources of the private Cart API can be identified:

- DynamoDB Table containing the carts
- DynamoDB Table containing the domain settings, i.e. if the products should be managed as SKU or as single entities
- DynamoDB Table possibly containing each product's settings
- An API Gateway providing APIs for both private methods (e.g. access to settings) and authenticated methods (adding items, retrieving cart, authentication)
- All the Lambda functions involved
- IAM roles needed by the AWS Resources for interacting with each-other

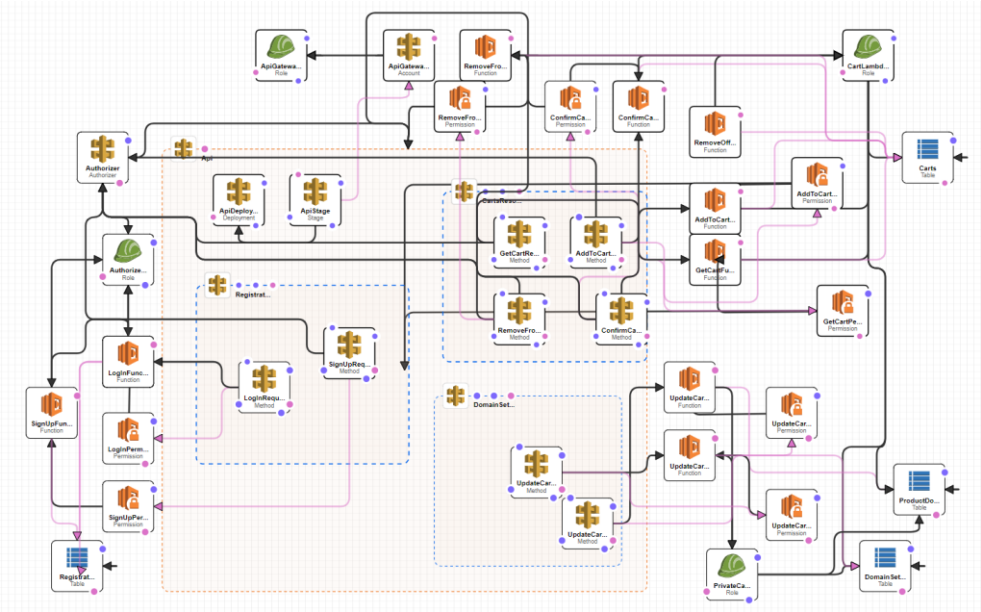


FIGURE 11 AWS CONSOLE SHOWING A GRAPHICAL VIEW OF THE CLOUDFORMATION TEMPLATE DESCRIBING AND PROVISIONING ALL THE AWS COMPONENTS NEEDED TO DEPLOY THE ARCHITECTURE PROVIDING A SUBSCRIBED DOMAIN'S CART APIS

The configuration template containing the information needed by CloudFormation to create this stack automatically is stored into an appropriate AWS S3 bucket. In order to invoke the generation of the stack, an explicit *CreateStack* command must be prompted by using CloudFormation APIs. Because of this, UBS publishes and manages some API to register the domain to obtain some Cart APIs: when a new domain is registered, the Lambda function that handles the signup contextually invokes the creation of a stack, customized for the domain; in fact, the created stacks, right after the creation, only differ for their resources' names,

customized by using the domain name passed as parameter to the template. The stack identifier, *StackId*, returned by the asynchronous call to *CreateStack* is associated with the domain.

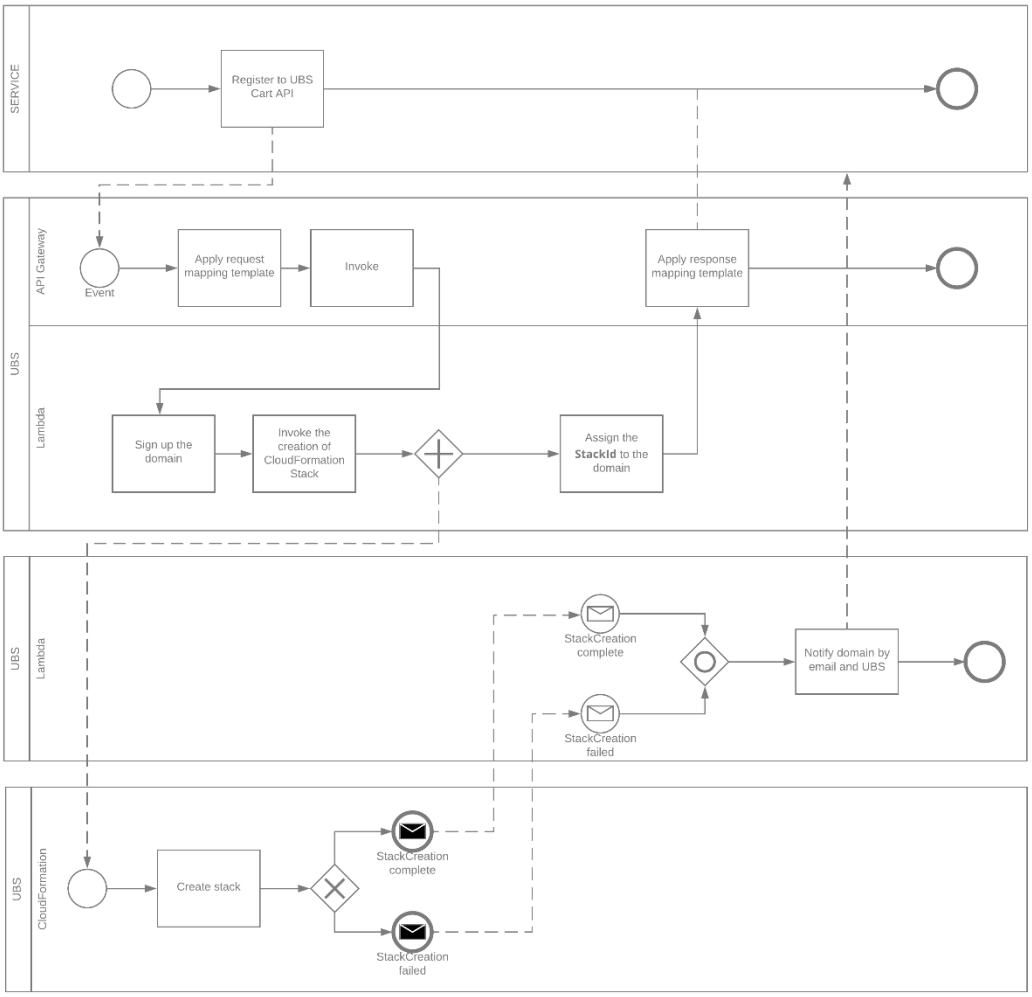


FIGURE 12 BPMN REPRESENTING THE PROCESS OF THE AUTOMATIC CREATION OF A DOMAIN'S CART API BY REGISTERING TO UBS AND USING CLOUDFORMATION TEMPLATE

Then, when the creation of the stack finishes, either because it completes or because it fails, an event is published to an SNS topic (that has been set to be notified for the stack's creation events) and a Lambda function is invoked to terminate the association between the domain and the stack, and to send the appropriate email notifications.

The main problem with this approach, that was present in the previously presented one too, is the lack of integration between the service's authentication and the Cart APIs. In fact, to be able to interact with these APIs, a client needs to authenticate to them. Like with the other approach, signing up and logging in can be integrated with the service's one, so that when a user logs in the service, the service logs him in the Cart APIs too. Then, based on how the APIs

are integrated into the service, either it caches the Cart authentication token or returns it to the client. In fact, if the client directly interacts with the Cart, it needs to provide the authentication token obtained during the authentication process; otherwise, if the service shows some APIs to handle the cart only acting as a proxy to the Cart APIs, it can either inject into the request the token associated with the client or make it passthrough if it had been previously provided to the client.

When the client needs to interact with the Cart API, considering the service showing proxy APIs, if the service is the one who owns the authentication token, it needs to inject it on all the requests.

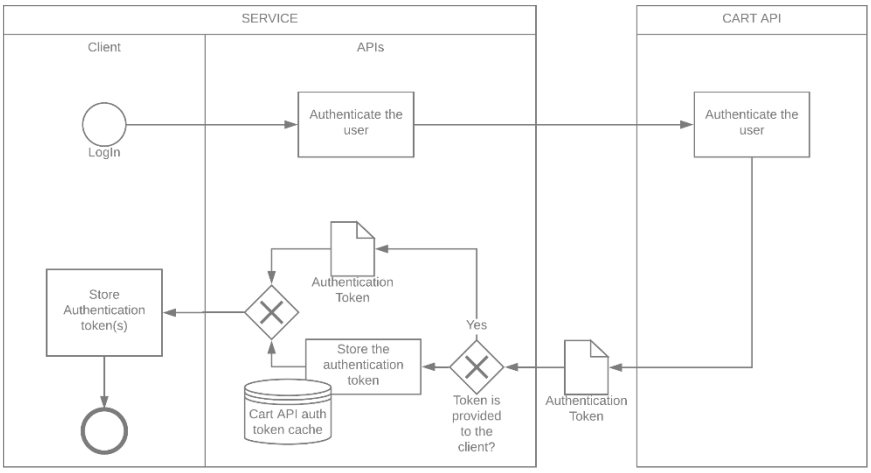


FIGURE 13 BPMN REPRESENTING THE PROCESS OF LOG-IN TO CART API FOR A SUBSCRIBED DOMAIN'S USER

This approach allows for a fine-grained cost analysis and so UBS can correctly invoice by usage the 3rd party services that make use of CART APIs.

4.6 UBS API

UBS defines some APIs to allow the users access the methods of the system. Most of them are RESTful APIs. Only reviews related APIs are built on GraphQL applications running on AWS AppSync.

4.6.1 RESTful APIs

For the RESTful APIs, a resource per Bounded Context has been defined. Shipments represents an exception, in fact part of its APIs is exposed into a sub-resource of the Orders resource.

The schema looks like this:

4.6.1.1 Book

To make Book context's store only useful information, that is, only on books put on sale, the database needs to be internally populated when a user wants to sell a book. When a new offer is being created, UBS takes care of inserting into the database information on the book being put on sale, if missing. This means that the creation of an item must not be a publicly available action: there is nobody that can *create* a book, meaning putting information of a book onto the system, nor *update* it or *delete* it. The only public action is *read*.

So, APIs of this context only consist of a read method. Being Book's APIs made available through HTTP RESTful protocol, a **/books** resource exists where the only exposed method is *GET*, requiring an **isbn** query string parameter by which the book whose information the requester wants to obtain specify it.

4.6.1.2 Cart

The API exposed on the **/carts** resource refer to the Public Cart API described previously described (cf. Public Cart API – Part 1).
Adding an item to a cart and removing it are authorized methods mapped respectively to the PUT and DELETE HTTP methods, requiring a *product* identifier. To retrieve the current content of the cart, an authorized call to GET method needs to be done, without the need to pass any query string parameter.
Sub-resources **/domain-settings**, **/login** and **/registration** handle all the rest of the Cart API, already discussed in depth.

4.6.1.3 Offer

To manage the users' offers, API have been designed and exposed on the **/offers** HTTP end-point. This is a straight-forward CRUD REST resource, where the creation is assigned to the POST method, the read to the GET method, the update to the PUT method and the deletion to the DELETE method. The *offer* identifier needs to be passed in order to apply any change or retrieve the offer, and it is automatically assigned on the creation.

4.6.1.4 Order

The order's lifecycle is not managed at all by the user. By using the APIs an order can only be read by making a GET HTTP request to **/orders**: in fact, it is created automatically by the system, updated through some actions internal to the service, and never deleted.

4.6.1.5 Shipment

On the **/shipments** HTTP resource they are exposed the methods that allow both the couriers and the customers to exchange information about the orders' shipments. The only exposed

method on the resource is GET: an HTTP call to this end-point requires the *shipment* identifier and returns all the information, including the tracking updates, relative to that shipment.
Then, various sub-resources are contained into /shipments:

- **/assignments**, where authorized REST CRUD requests can be made by the courier to manage the pending and assigned assignments
- **/couriers**, where a courier can update its own information by making a PUT request
- **/login** and **/registration** to signup and login by using POST requests

4.6.1.6 User

At the actual state of UBS, there is no need for a user to know information on other users. So, there is no API that exposes it: this context's resource, **/users**, only provides a user information on his credit through HTTP GET and allows a user to sign up and log in by making POST requests to sub-resources **/login** and **/registration**.

4.7 The projects' structures

There are a couple structures worth describing in an AWS serverless project: the AWS components running the service, and the code running on the Lambda functions.
On AWS, UBS is made of

- 52 Lambda Functions
- 26 IAM Roles
- 15 DynamoDB Table
- 5 SNS Topics
- 3 Step Functions
- 2 CloudFormation templates
- 1 S3 Bucket
- 1 AppSync API
- 1 SES domain
- 1 API Gateway, consisting of 16 resources, 25 methods, 3 authorizers and 1 stage

The 52 Lambda functions (ordered by sanitized name) are divided like this:

Sanitized name	Env.	BOOK	CART	OFFER	ORDER	SHIPMENT	USER
AcceptAssignment	Node.js 8.10					●	
AddToCart	Java 8		●				
AssignParcelToCourier	Node.js 8.10					●	
AssignShippingTrackingOrder	Java 8				●		
AuthorizeBookReviews	Node.js 8.10	●					
CancelPendingOrder	Java 8				●		
CartCheckToken	Java 8		●				
CartLogIn	Java 8		●				
CartSignUp	Java 8		●				
CheckAndLockCredit	Java 8						●
CheckAndLockOffers	Java 8			●			
CheckToken	Java 8						●
ChooseCourierForParcel	Node.js 8.10					●	
CloseBecauseDeliveredOrder	Java 8				●		
CompleteUserRegistration	Java 8						●
ConfirmCart	Java 8		●				
ConfirmLockOffers	Java 8			●			
ConfirmSpentCredit	Java 8						●
CourierCheckToken	Node.js 8.10					●	
CourierLogIn	Node.js 8.10					●	
CourierSignUp	Node.js 8.10					●	
CreateBook	Java 8	●					
CreateOffer	Java 8			●			
CreatePendingOrder	Java 8				●		
EmptyUserCart	Java 8		●				
FillCredit	Java 8						●
GetAssignmentStatus	Node.js 8.10					●	
GetBook	Java 8	●					
GetCart	Java 8		●				
GetCredit	Java 8						●
GetOffers	Java 8			●			
GetOrder	Java 8				●		
GetPendingAssignments	Node.js 8.10					●	
GetRunningShippings	Node.js 8.10					●	
GetShippingTracking	Node.js 8.10					●	
InvokeAssignmentStep	Node.js 8.10					●	
LogIn	Java 8						●
NewCourierRegistered	Node.js 8.10					●	
NotifyAllCouriersForAssignment	Node.js 8.10					●	
NotifyAssignment	Node.js 8.10					●	
PutBook	Java 8	●					
RefuseAssignment	Node.js 8.10					●	
RemoveFromCart	Java 8		●				
RemoveOffersFromCarts	Java 8		●				
RequestConfirmationOrder	Java 8				●		
SignUp	Java 8						●
UnlockCredit	Java 8						●
UnlockOffers	Java 8			●			
UpdateCartDomainProductSettings	Java 8		●				
UpdateCartDomainSettings	Java 8		●				
UpdateCourierInfo	Node.js 8.10					●	
UpdateTracking	Node.js 8.10					●	

Most of the functions are written in Java; Courier Context functions are written in Node.js, instead. It is also obviously possible to use several languages to write functions belonging to the same context.

For the functions written Java a single Java gradle project exists. Nevertheless, because of the nature of the architecture, it is actually divided in modules, consisting of a general *utils* module and an additional module per context.

To help with the deployment, each of these gradle modules contain some gradle tasks that allow the creation of a deployment package (a zip file containing all the classes), its upload on Amazon S3 and finally the update of the Lambda functions of the context (rf. Appendix).

4.8 Java vs Node.js: Log In functions comparison

In both languages I have written a function to make the user log in. They are more or less equivalent in what they do, so it is nice to compare their performances.

The test consists on the invocation of the function to log in the same user several times. To test it in depth, various configurations have been considered:

- 300 times using 1 thread
- 300 times using 5 parallel threads
- 300 times using 10 parallel threads
- 300 times using 15 parallel threads
- 300 times using 20 parallel threads

Each test interleaved by 15 minutes. Then, to test the cold start deeper (more on this later), I have reduced the interval between groups of calls by 2 minutes at a time starting from 12, and invoking:

- 60 times using 1 thread
- 60 times using 5 threads
- 60 times using 10 threads
- 60 times using 15 threads
- 60 times using 20 threads

To avoid having the tests being influenced by API Gateway, I used AWS command line interface to invoke functions, by using a Java Application for automation.

The first thing that can be noticed is how faster node.js is in processing the invocation under any circumstances: the overall average invocation's durations in my tests where 409.11ms for Java, 28.75ms for node.js.

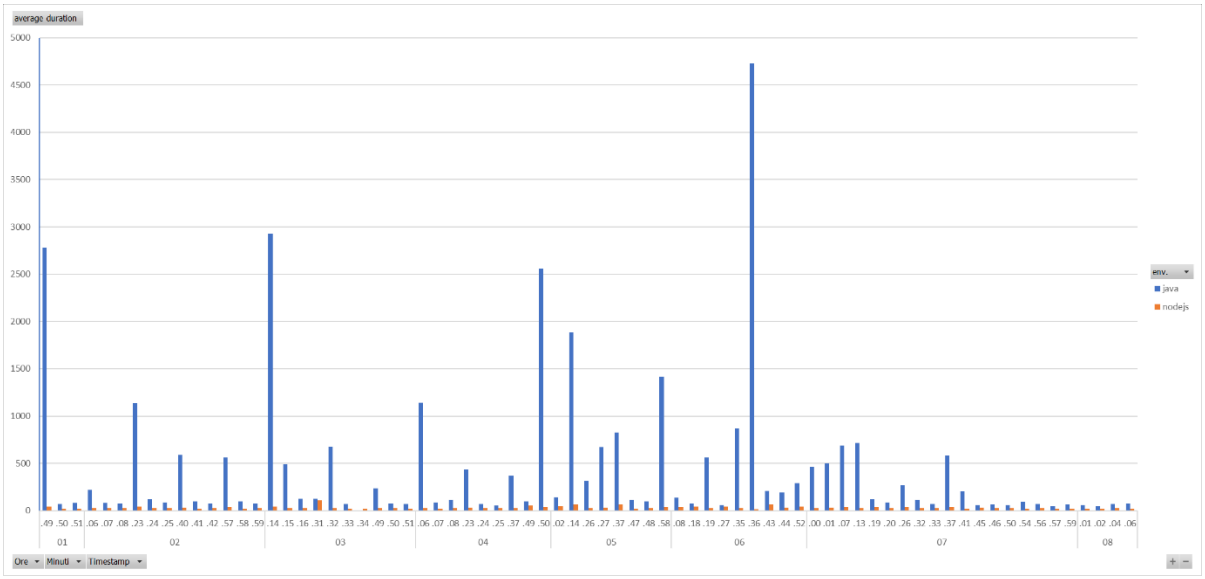


FIGURE 14 AVERAGE AWS LAMBDA INVOCATIONS' DURATION WITH JAVA (BLUE) AND NODE.JS (ORANGE)

This chart compares the average duration of invocations by minute. Zooming in on Y axis, this is the result.

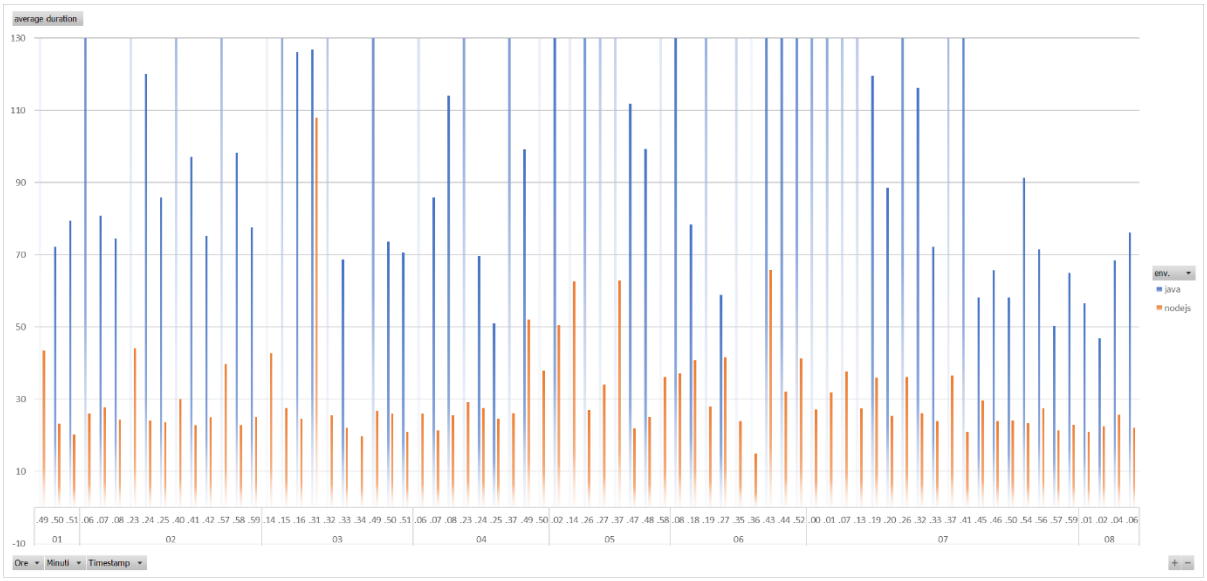


FIGURE 15 AVERAGE AWS LAMBDA INVOCATIONS' DURATION WITH JAVA (BLUE) AND NODE.JS (ORANGE) CAPPED TO 130MS

In these two charts it can be noticed that there are apparently strange spikes in the Java environment and analysing the data they can be linked to the first invocations of each group. When reducing the domain, for example considering only the first group of tests (300 invocations in a row), that spans about 2 minutes for both the environments, it is clearer: the first Java invocations are very slow respect to the subsequent ones. Also Node.js instances are slower at first and then much faster, but the ratio between the duration times is not comparable.

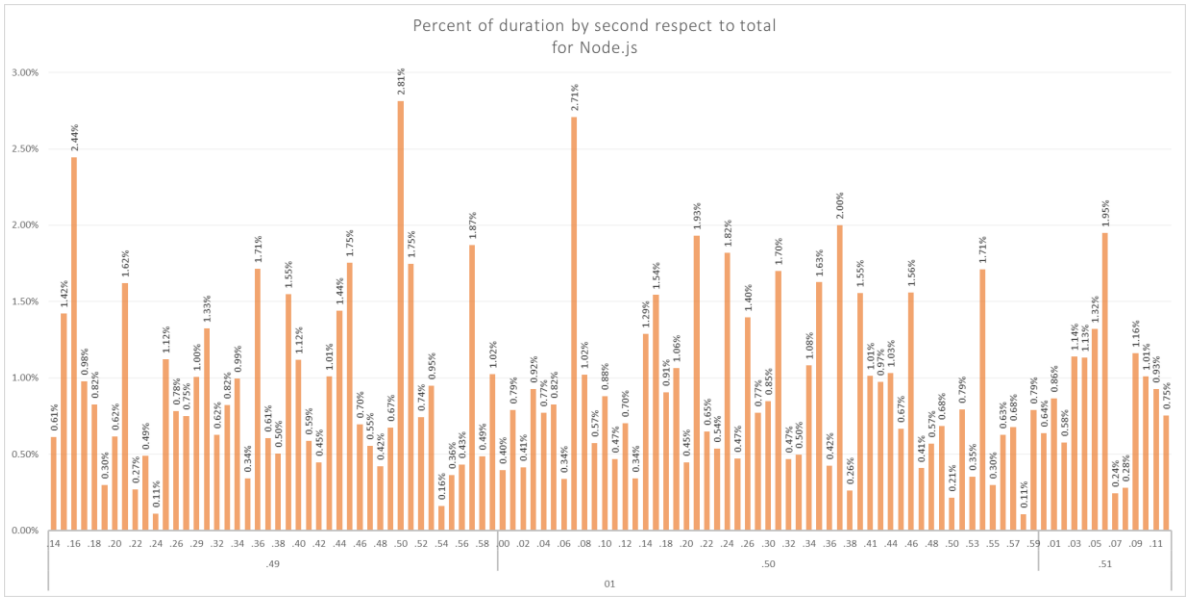


FIGURE 16 DISTRIBUTION OF THE DURATION TIME BY SECOND IN NODE.JS – 300 REQUESTS ON 1 THREAD

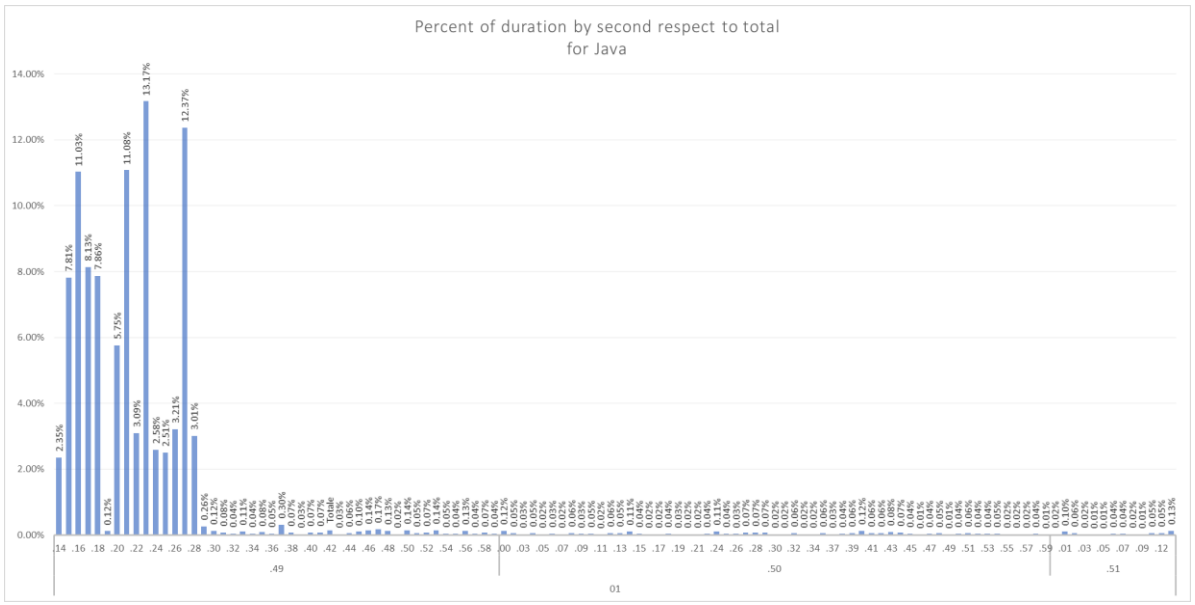


FIGURE 17 DISTRIBUTION OF THE DURATION TIME BY SECOND IN JAVA - 300 REQUESTS ON 1 THREAD

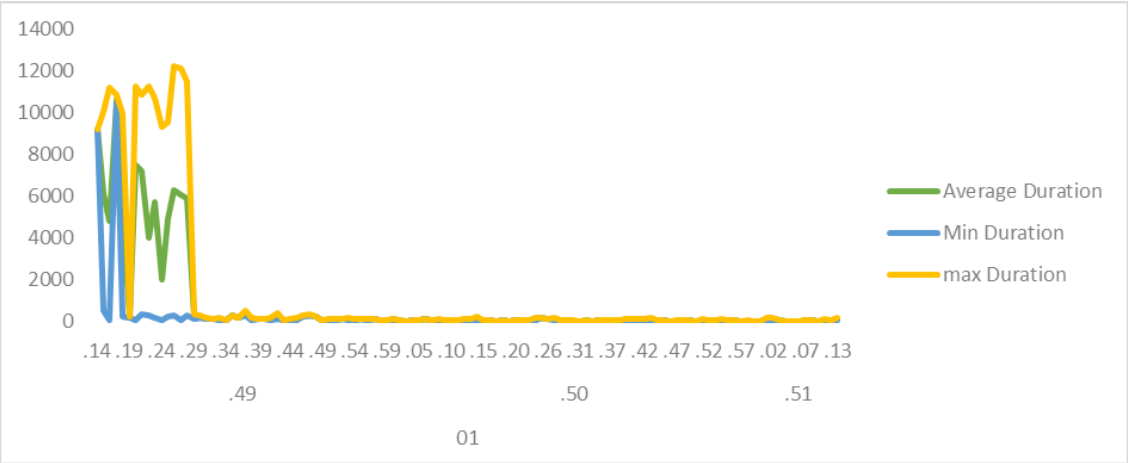


FIGURE 18 AVERAGE (GREEN), MINIMUM(BLUE) AND MAXIMUM(ORANGE) AWS LAMBDA INVOCATIONS' DURATION WITH JAVA FOR 300 LOG-IN

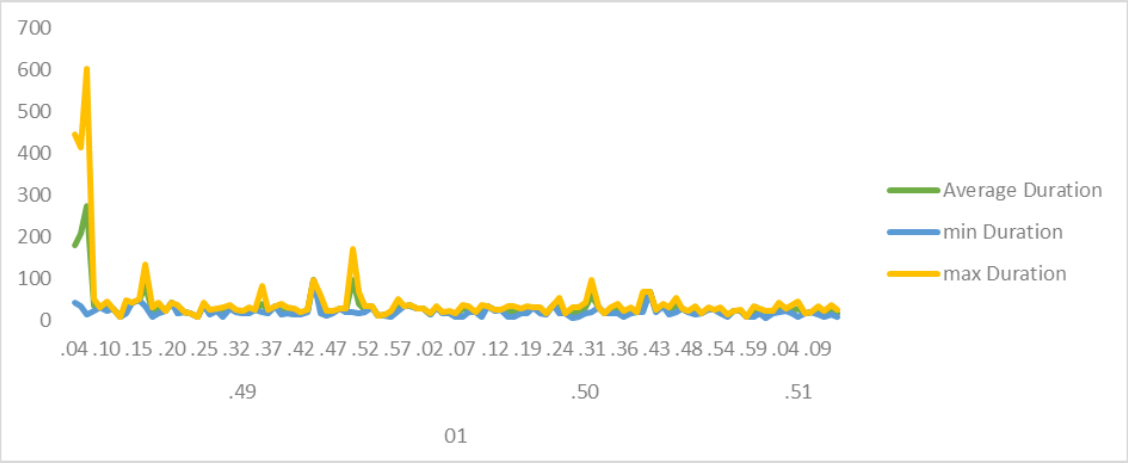


FIGURE 19 AVERAGE (GREEN), MINIMUM(BLUE) AND MAXIMUM(ORANGE) AWS LAMBDA INVOCATIONS' DURATION WITH NODE.JS FOR 300 LOG-IN

Ignoring the first seconds, the two environments behave comparably, even though Node.js keeps looking much faster.

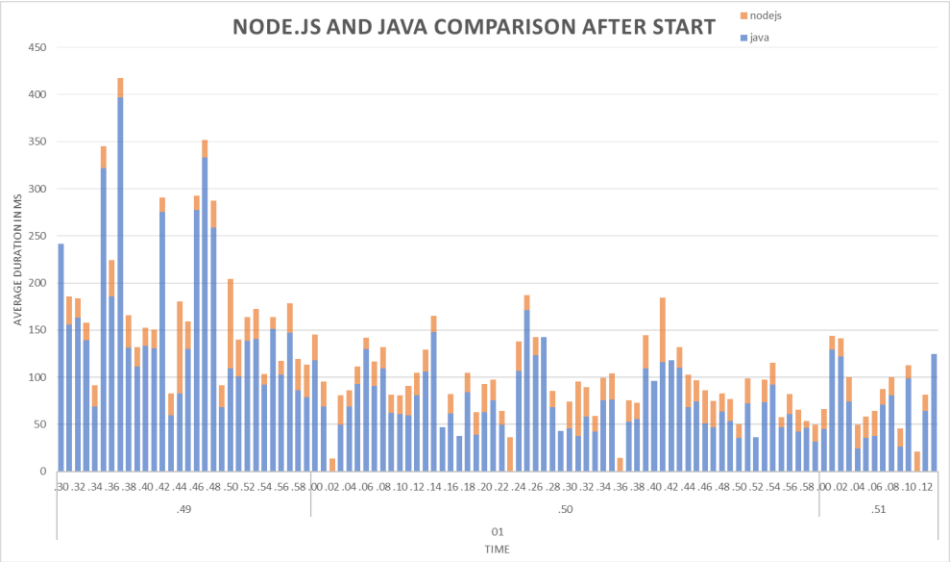


FIGURE 20 AVERAGE AWS LAMBDA INVOCATIONS' DURATION WITH JAVA (BLUE) AND NODE.JS (ORANGE) WITHOUT COLD START

The problem, in fact, is **cold start**, that is, the time that the environment the function runs on needs to prepare.

Various experiments I have read (1[6], 2[7], 3[8]) and the ones I have personally run demonstrate how the cold start of a Lambda function depends mostly on:

- Language
- Resources assigned to the function
- Idle time

It is hypothesized that it also depends on the AWS server and the time of day.

So, when a Lambda function has been idle for some time, the next invocation will suffer a lag that will make it slower. What also happens is that if the Lambda function is allowed to run in many parallel instances, each of them will experience this lag.

The following two charts show how the total and the maximum duration of the executions changes varying the number of parallel requests and the idle time before the first call.

6 https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810

7 https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76

8 https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f

These tests were performed grouping the requests in 15 groups consisting of 60 calls, varying the number of parallel requests from 1 to 20 and decreasing the idle time by 2 minutes per group, starting from 12.

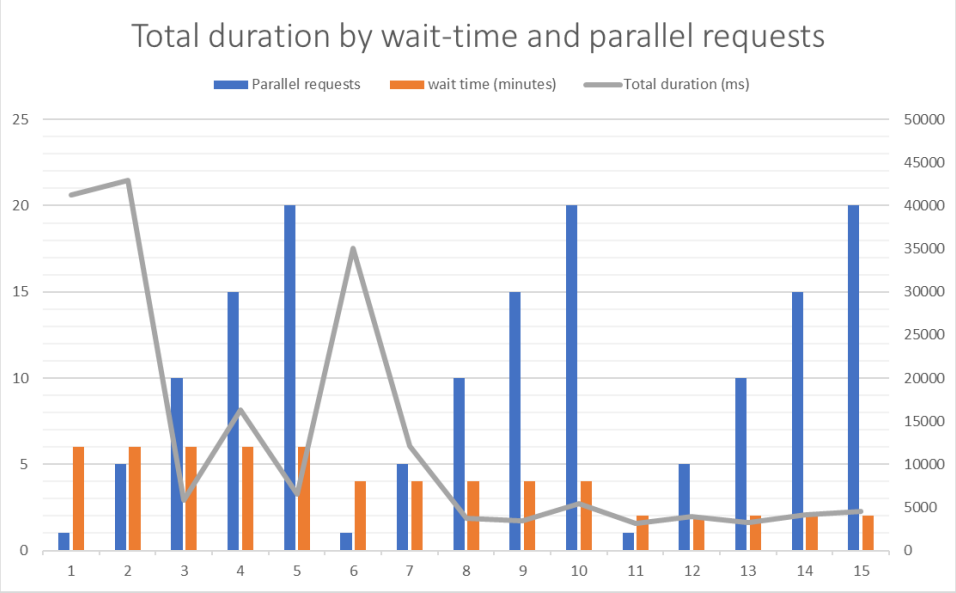


FIGURE 21 TOTAL AWS LAMBDA INVOCATIONS' DURATION WITH JAVA FOR 900 LOG-IN REQUESTS DIVIDED IN 15 GROUPS VARYING THE PARALLEL REQUESTS COUNT AND THE WAIT TIME BETWEEN REQUESTS

As the wait-time between the calls, that is, the time Lambda function spends idle, decreases to what in my experiments was 2 minutes, the need for a cold start disappeared and probably the same JVM that was previously running some instances was reused.

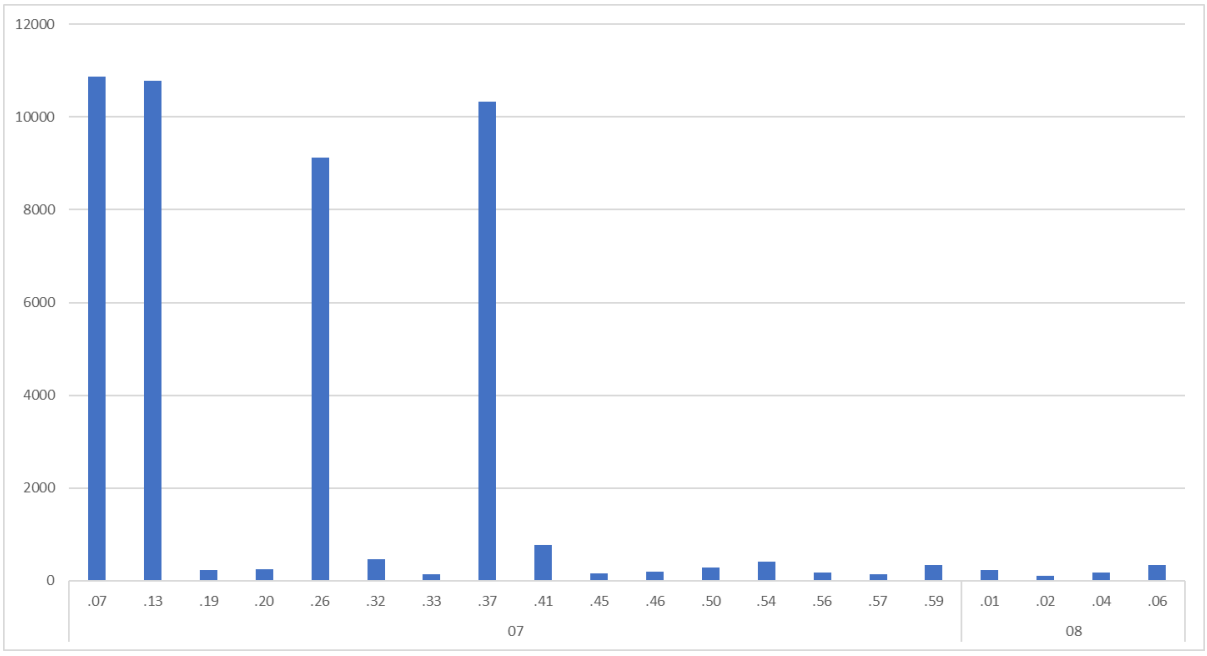


FIGURE 22 MAXIMUM AWS LAMBDA INVOCATIONS' DURATION WITH JAVA FOR 900 LOG-IN REQUESTS DIVIDED IN 15 GROUPS VARYING THE PARALLEL REQUESTS COUNT AND THE WAIT TIME BETWEEN REQUESTS

4.8.1 Considerations

Considering only the results of these tests, the huge difference in cold start and the relevant time execution difference, the natural conclusion should be that Node.js is the absolute winner in the comparison and that there is no point in considering Java for writing functions. In fact, even though the latency could be considered a non-problem, on AWS time and cost walk together, and considering the increasing of the resources associated to a Lambda function is not a solution, because the cost increases with the resources too.

On the other hand, the differences in the language must be considered. Java has years of existence on its back, there are tons of frameworks and libraries and, leaving cold starts aside, it is consistent in performances and reliable.

Finally, personal preferences should be accounted for: I'm a long-time Java developer and it is easier for me to write in Java. Coming from Java you need to get used to JavaScript strange behaviours, the natural callback system and the total absence of typing.

5 Conclusions

In my opinion, Software engineering is about the design and not the code, and this is the reason why there is no code in all the preceding chapters and sections. The code can be found in the appendix after this chapter.

Nevertheless, the project this work accompanies can be effectively thought to as a made of two parts: design and code.

The development process model chosen to be used on an application built following a serverless architecture can definitely be the V-model.

The design part lives on several layers of the V-model, as it can be seen in this work, but also on different levels of detail of the architecture:

- First, it needs to be designed a separation of contexts inside the business process
- Each process inside the business needs to be associated with a flow of data that must be defined and designed, either living into a single Bounded Context or being cross-concern
- Each data flow can be designed in several ways and it is usually made of smaller single components
- For each component, the platform(s) the application is being developed on can grant a set of choices to choose from

Then it's time for the coding phase and the choices it comes with: the first choice stands on the line between the two phases, that is, the language to use to develop the function. Watch out, though: the development of the function as the final step is not always true, but only for the minor part of the final components, that are actually Lambda functions or their counterparts on platforms other than AWS. It needs to be highlighted and underlined that serverless is not only FaaS. As it has been proven during this work, for most of the business process, especially when no particular rules apply, properly configuring data access, exploiting

the provisioned authentication system and correctly writing the mapping templates between the components, can be enough to cover most of the application.

The design phase, in any case, also permeates the code phase. Coding without a clear design in mind, leaving behind, ignored, the design patterns that rule the development nowadays, can only lead to a bad work.

Anyway, these two parts make up to the most part of the job that needs to be done to build an application based on a serverless architecture, and they mostly deal with software, and in particular with software whose specific purpose is to run the business process. All the lower parts consisting of designing and developing the infrastructure, whether hardware or software, is left behind. This leads to a reduction in the team size, so much consistent that the prototypical application for this work, as incomplete, has been built by a single person. But this also leads to much more flexibility, and any cost due to over-provision of the hardware is totally removed.

About the cost, some estimations should be made. They can be divided in two parts: service provisioning costs and service development cost. Developing your own hardware and software infrastructure for provisioning your service faces lowly varying costs that hardly match the real need of the system, in fact some calculations should be made for deciding the hardware to be bought: it is not convenient to follow a trial and error approach for increasing the resources associated to the system as they are no (more) sufficient to provide the service, but on the same time over-provisioning should be avoided. Service provisioning costs do not stop here, but also deal with maintaining the infrastructure both up-and-running and working. On the other side, with a serverless approach, excluding a hardly used self-provisioned Apache OpenWhisk, the costs only deal with the work that the service performs, **mostly**. To be fair, an AWS hosted serverless application also faces some costs that are not very intuitive: I'm talking about DynamoDB costs related to the reserved capacity applied on each table and on each additional global secondary index, that if not properly tweaked and thought to in advance can cause useless high costs. Additionally, DynamoDB and Lambda functions' invocations might apparently look like they were the only costs of the entire application, leaving aside for a moment the additional data storage on S3, that is nowadays used regardless of the self-provisioned or AWS-hosted architecture. Truth is, API Gateway and AppSync come with a price that depends on the data coming out and on the amount of API invocations. The only free components are CloudFormation, IAM and Cognito, that do nothing alone but are

useful to integrate with the other components. Using Step Functions for orchestrating functions comes with a price too, but it is quite limited, and the appropriate usage of this AWS component leads to a low cost. SNS and SES for dispatching notifications or events comes with a price too, but it is quite easy not to exceed the free tier.
About the development costs, it is a declared goal for serverless architecture to cut its costs too, by allowing agile approach and continuous delivery. Each function can be independently deployed and tested and the problems isolated. On the other hand, the size of the codebase to maintain could increase remarkably if the design has some flaws.

For what concerns the coding phase, first of all it needs to be underlined that I have been a professional Java developer for some years before starting this work and before approaching JavaScript and Node.js. Nevertheless, I have hardly found a reason why not to advice to use Node.js over Java. It is more efficient in random calls; the web is full of well done tools and libraries; it does not need to be compiled, so the deployment during the final test phases is way quicker. What I liked most about Java during the development of the project is Gradle. A good plugin for automating the deployment process exists, and with some tweaking I have succeeded to totally automate the updating of the functions by zipping the deployment package, uploading it to S3 and using it to update the codebase of the Lambda functions I wanted to. Being an interpreted language, Node.js does not need to compile the files the developers write, and so a powerful tool like Gradle is not required; to obtain similar results for automating the deployment, simple scripts, depending on the OS used, can be written and launched, thanks to the powerful command line interface AWS provides.

Some considerations on the area of usage of the Lambda functions should be made too: as previously mentioned in some occasions, building APIs is not strictly related to writing Lambda Functions; for most cases, the usage of API Gateway or AppSync is enough to fulfil the goal. AWS Lambda is a compute service: its purpose is to compute data, transforming the provided input or using it to trigger a transformation to data stored somewhere else. This service should be used for complex actions whose application is required frequently and that can be related to events. Using it to build RESTful APIs is an overkill and leads to sub-optimal results.

This might be not widely understood, in fact it is easy to find on the web several people opposing to this approach, evaluating it not cost-wise, requiring too much effort for the results and sometimes even considering it a bait.
The poor reception is limited, anyway, but it has to be considered. Another symptom to this is the low presence of questions posted on StackOverflow with tags related to FaaS. This can be either because the development of functions on the various platform is very straight-forward, or because they are not widely used, or because the general level of the developers exploiting this approach is medium-high. I vote for the last two options, considering the level of questions that I read every day visiting StackOverflow, and that I have explored most of the official documentation on the AWS components that I have used.

The imperfection of the official docs and the low presence of these new AWS components on StackOverflow (EC2 is very tagged, on the other side) have been one of the difficulties I have met during the various phases of the project. Both on pure design and conceptual ideas and during the implementation phase, some doubts were raised that I have hardly resolved by myself after both research and trial-and-error, and that earned me some achievements on StackOverflow as a side-effect.

Serverless is not only FaaS, and I would like to point it out one more time. Apart from the previously mentioned ways to deploy a serverless architecture barely built on functions, there is Backend as a service. It is way more used, at present day, than FaaS thanks to Google's Firebase and to how much it fits with everyday mobile applications, often requiring poor logics and real time database access without latency or entities staying in the middle between the client consuming and producing data and the data itself. A good client-side SDK, and perfect and invisible automatisms for handling data storage and authentication are the keys of its success.

In conclusion, I think that serverless is far from perfect, but the competition that the giants are living and the increasing adoption by the customers can only improve this approach, that, and I would like to underline it once more, should not be chosen blindly but opportunely tailored on the needs the service has, measuring its advantages and disadvantages.

6 References

12factor. (n.d.). *The twelve-factor app.* Retrieved from 12factor: https://12factor.net/

Amazon AWS. (n.d.). *Amazon AWS*. Retrieved from Amazon AWS: aws.amazon.com

Amazon AWS. (n.d.). *AWS Documentation.* Retrieved from Amazon AWS: https://docs.aws.amazon.com/

Amazon AWS. (n.d.). *Netflix & AWS Lambda Case Study.* Retrieved from Amazon AWS: https://aws.amazon.com/it/solutions/case-studies/netflix-and-aws-lambda/

Amazon AWS. (n.d.). *Serverless Computing and Applications.* Retrieved from Amazon AWS: https://aws.amazon.com/serverless

Anonymous. (2012, May 15). *Why REST Keeps Me Up At Night.* Retrieved from Programmable Web: https://www.programmableweb.com/news/why-rest-keeps-me-night/2012/05/15

Apache OpenWhisk. (n.d.). *What Is Serverless Computing, and Why Should I Care?* Retrieved from Apache OpenWhisk: https://openwhisk.apache.org/serverless

Barr, J. (2013, July 1). *Transaction library for DynamoDB.* Retrieved from Amazon AWS: https://aws.amazon.com/it/blogs/aws/dynamodb-transaction-library/

Bennage, C. (2017, February 28). *Command and Query Responsibility Segregation (CQRS) pattern.* Retrieved from GitHub: https://github.com/mspnp/architecture-center/blob/master/docs/patterns/cqrs.md

Buliani, S. (2016, February 26). *Using Amazon API Gateway as a proxy for DynamoDB.* Retrieved from Amazon AWS: https://aws.amazon.com/it/blogs/compute/using-amazon-api-gateway-as-a-proxy-for-dynamodb/

Can, S. (2017, November 28). *4 Advantages of using Java with AWS Lambda.* Retrieved from OpsGenie: https://engineering.opsgenie.com/4-advantages-of-using-java-with-aws-lambda-21c0dc539de3

Chaubey, M. (2018, January 12). *Scale Cube: Simplified Scale Model for Microservices.* Retrieved from DZone: https://dzone.com/articles/scale-cube-simplified-scale-model

Chris Richardson, F. S. (2016). *Microservices, from Design to Deployment.* NGINX.

Christopher Bennage, V. B. (2017, February 28). *Event Sourcing pattern.* Retrieved from GitHub: https://github.com/mspnp/architecture-center/blob/master/docs/patterns/event-sourcing.md

Cockburn, A. (2005). *Ports and Adapters*. Retrieved from Alistair Cockburn Wiki: http://wiki.c2.com/?PortsAndAdaptersArchitecture

Cockburn, A. (2014, November 4). *Hexagonal Architecture.* Retrieved from Alistair Cockburn: http://wiki.c2.com//?HexagonalArchitecture

Conway, M. E. (1968). How do Committees Invent? *Datamation*, 28-31.

Cui, Y. (2016, January 16). *I'm afraid you're thinking about AWS Lambda cold starts all wrong*. Retrieved from Hackernoon: https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f

Cui, Y. (2017, July 18). *Applying the Saga pattern with AWS Lambda and Step Functions.* Retrieved from theburningmonk.com: https://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

Dias, A. L. (2017, December 1). *Microservices Design Patterns - API Gateway.* Retrieved from DZone: https://dzone.com/articles/7-important-microservices-design-patterns-every-de

Evans, E. (2003). *Domain-Driven Design.* Addison Wesley.

Fowler, M. (2018, May 22). *Serverless Architectures.* Retrieved from Martin Fowler: https://martinfowler.com/articles/serverless.html

Gilham, I. (2016, March 22). *Triggering a Lambda from SNS using CloudFormation*. Retrieved from iangilham.com: https://iangilham.com/2016/03/22/Sns-trigger-lambda-via-cloudformation.html

Graca, H. (2017, November 16). *DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together.* Retrieved from Herberto Graca: https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/

GraphQL. (n.d.). *GraphQL.* Retrieved from GraphQL: https://graphql.org/

Helfer, J. (2016, March 14). *A guide to authentication in GraphQL.* Retrieved from Apollo: https://dev-blog.apollodata.com/a-guide-to-authentication-in-graphql-e002a4039d1

Huston, T. (n.d.). *What is microservices architecture?* Retrieved from SmartBear: https://smartbear.com/learn/api-design/what-are-microservices/

James. (2018, March 9). *Azure Functions – Significant Improvements in HTTP Trigger Scaling.* Retrieved from Azure Trenches: https://www.azurefromthetrenches.com/azure-functions-significant-improvements-in-http-trigger-scaling/

James Lewis, M. F. (2014, March 25). *Microservices.* Retrieved from Martin Fowler: https://martinfowler.com/articles/microservices.html

jwt. (n.d.). *jwt.* Retrieved from jwt.io: https://jwt.io/

Kumar, D. (2017, May 18). *How to write GraphQL Apps using AWS Lambda*. Retrieved from Cloud Academy: https://cloudacademy.com/blog/how-to-write-graphql-apps-using-aws-lambda/

Martin L. Abbott, M. T. (2009). *The Art of Scalability.*

Martin, R. C. (1996). *Object Oriented Design Quality Metrics: an analysis of dependencies.*

Maruti TECHLABS. (2017, May 4). *What is Serverless Architecture? What are its criticisms and drawbacks?* Retrieved from Medium: https://medium.com/@MarutiTech/what-is-serverless-architecture-what-are-its-criticisms-and-drawbacks-928659f9899a

Maruti TECHLABS. (n.d.). *SERVERLESS ARCHITECTURE THE FUTURE OF BUSINESS COMPUTING.* Retrieved from Maruti TECHLABS: https://www.marutitech.com/serverless-architecture-business-computing/

Masashi Narumoto, G. C. (2017, June 23). *Compensating Transaction pattern.* Retrieved from Microsoft Azure: https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction

Matt McLarty, I. (2016, June 9). *Learn from SOA: 5 lessons for the microservices era.* Retrieved from InfoWorld: https://www.infoworld.com/article/3080611/application-development/learning-from-soa-5-lessons-for-the-microservices-era.html

Mauro, T. (2015, February 19). *Adopting Microservices at Netflix: Lessons for Architectural Design.* Retrieved from NGINX: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

Mauro, T. (2015, February 19). *Adopting Microservices at Netflix: Lessons for Architectural Design.* Retrieved from NGINX: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

Merson, P. (2015, November 12). *Transaction across REST microservices.* Retrieved from StackOverflow: https://stackoverflow.com/questions/30213456/transactions-across-rest-microservices

Mytton, D. (2017, August 10). *Who has the serverless advantage?* Retrieved from A cloud guru: https://read.acloud.guru/aws-lambda-vs-google-cloud-functions-vs-azure-functions-who-has-the-serverless-advantage-f6c2535e72f4

Nommensen, P. (2015, February 6). *It's Time to Move to a Four-Tier Application Architecture*. Retrieved from NGINX: https://www.nginx.com/blog/time-to-move-to-a-four-tier-application-architecture/

Palermo, J. (2008, July 29). *The Onion Architecture*. Retrieved from Jeffrey Palermo: http://jeffreypalermo.com/blog/the-onion-architecture-part-1/

Pant, H. (2018, April 5). *A brief history of serverless (or, how I learned to stop worrying and start loving the cloud).* Retrieved from freeCodeCamp: https://medium.freecodecamp.org/a-brief-history-of-serverless-or-how-i-learned-to-stop-worrying-and-start-loving-the-cloud-7e2fc633310d

Riady, Y. (2017, September 3). *Serverless Authentication with JSON Web Tokens*. Retrieved from yos.io: https://yos.io/2017/09/03/serverless-authentication-with-jwt/

Richardson, C. (2018). *Microservices Patterns.* Manning.

Richardson, C. (n.d.). *microservices.io*. Retrieved from microservices.io: http://microservices.io

Riggins, J. (2015, July 10). *How To Design Great APIs With API-First Design.* Retrieved from Programmable Web: https://www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10

Samohkin, V. (2018, January 6). *DDD Strategic Patterns: How to Define Bounded Contexts.* Retrieved from DZone: https://dzone.com/articles/ddd-strategic-patterns-how-to-define-bounded-conte

serverless. (n.d.). *serverless*. Retrieved from serverless.com: https://serverless.com/

Severson, M. (2016, August 17). *Introduction to CloudFormation for API Gateway*. Retrieved from jayway: https://blog.jayway.com/2016/08/17/introduction-to-cloudformation-for-api-gateway/

Stenberg, J. (2014, October 30). *Domain-Driven Design with Onion Architecture.* Retrieved from InfoQ: https://www.infoq.com/news/2014/10/ddd-onion-architecture

Stenberg, J. (2014, October 31). *Exploring the Hexagonal Architecture.* Retrieved from InfoQ: https://www.infoq.com/news/2014/10/exploring-hexagonal-architecture

Stubailo, S. (2017, June 27). *GraphQL vs. REST.* Retrieved from Apollo: https://dev-blog.apollodata.com/graphql-vs-rest-5d425123e34b

Threlkeld, R. (2018, April 24). *GraphQL authorization with multiple data sources using AWS AppSync.* Retrieved from Hackernoon: https://hackernoon.com/graphql-authorization-with-multiple-data-sources-using-aws-appsync-dfae2e350bf2

Wikipedia. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/

Wittig, M. (2016, August 5). *API Gateway Custom Authorization with Lambda, DynamoDB and CloudFormation*. Retrieved from cloudonaut.io: https://cloudonaut.io/api-gateway-custom-authorization-with-lambda-dynamodb-and-cloudformation/

Woods, D. (2015, July 30). *Microservices: the right way.* Retrieved from Slide Share: https://www.slideshare.net/danveloper/microservices-the-right-way

Yigal, A. (2017, August 8). *AWS Lambda vs. Azure Functions vs. Google Functions.* Retrieved from logz.io: https://logz.io/blog/serverless-guide/

Young, G. (2010). *CQRS Documents.*

Ziemoński, G. (2017, February 27). *Onion Architecture is interesting.* Retrieved from DZone: https://dzone.com/articles/onion-architecture-is-interesting

Ziemoński, G. (2017, January 17). *Perfecting Your SOLID Meal With DIP.* Retrieved from DZone: https://dzone.com/articles/perfecting-your-solid-meal-with-dip

7 Appendix

7.1 Code

7.1.1 Java projects gradle

7.1.1.1 Root project build.gradle

Gradle build script containing all the scripts to build, create the deployment package, update it to S3 and call the individual projects' script to update the AWS Lambda function code.

```
import jp.classmethod.aws.gradle.lambda.AWSLambdaUpdateFunctionCodeTask
import jp.classmethod.aws.gradle.lambda.S3File
import jp.classmethod.aws.gradle.s3.AmazonS3FileUploadTask
import jp.classmethod.aws.gradle.s3.CreateBucketTask

apply plugin: 'java'
group 'balsick-aws'
def version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

buildscript {
    repositories {
        maven { url 'https://repo.spring.io/plugins-snapshot' }
        maven { url "https://plugins.gradle.org/m2/" }
    }
    dependencies {
        classpath 'io.spring.gradle:dependency-management-plugin:1.0.3.RELEASE'
        classpath "jp.classmethod.aws:gradle-aws-plugin:0.+"
    }
}
apply plugin: "io.spring.dependency-management"

task buildAllZips {
    subprojects {
        apply plugin: 'java'
        task buildZip(type: Zip) {
            task ->
                destinationDir = file("${rootProject.projectDir}/build/distributions")
                println "${project.name}"
```

```
            from task.project.compileJava
            from task.project.processResources
            into('lib') {
               from task.project.configurations.runtimeClasspath
            }
      }
   }
}

def myBucketName = 'it.balsick.cloud.aws.thesis.lambdafunctions'
def regionName = 'eu-west-1'

task uploadAllToS3 {
   subprojects {
      apply plugin: 'jp.classmethod.aws'
      aws {
         profileName = "s3"
         region = regionName
      }
      apply plugin: "jp.classmethod.aws.s3"
      apply plugin: "io.spring.dependency-management"

      task updateToS3(type: AmazonS3FileUploadTask, dependsOn: buildZip) {
         task ->
            bucketName myBucketName
            overwrite true
            def zipName = "${task.project.name}-${version}.zip";
            println "Zip file name:\t${zipName}"
            println "S3 uploading file: ${rootProject.projectDir}/build/distributions/${zipName}"
            file file("${rootProject.projectDir}/build/distributions/${zipName}")
            key zipName
      }
   }
}

apply plugin: "jp.classmethod.aws.s3"

task createBucket(type: CreateBucketTask) {
   bucketName myBucketName
   region regionName
   ifNotExists true
}

ext.updateLambdaFunctionsTask = { Project prj, List<String> functions ->
   apply plugin: "jp.classmethod.aws.lambda"
   lambda {
      region = regionName
   }
   functions.each {
      s ->   task "updLambda-${prj.name}-${s}"(type: AWSLambdaUpdateFunctionCodeTask, dependsOn:
uploadAllToS3) {
            functionName = s
            s3File = new S3File()
            s3File.bucketName = myBucketName
            s3File.key = "${prj.name}-${version}.zip"
         }
   }
}
```

||

7.1.1.2 Utils build.gradle

Gradle build script for the Utils java project containing the dependencies that the other projects use too.

```
group 'balsick-aws'
version '1.0-SNAPSHOT'

apply plugin: 'java-library'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

buildscript {
    repositories {
        maven { url 'https://repo.spring.io/plugins-snapshot' }
    }
    dependencies {
        classpath 'io.spring.gradle:dependency-management-plugin:1.0.3.RELEASE'
    }
}

apply plugin: "io.spring.dependency-management"

dependencyManagement {
    imports {
        mavenBom 'com.amazonaws:aws-java-sdk-bom:1.11.228'
    }
}

dependencies {
    api 'com.amazonaws:aws-lambda-java-core:1.1.0'
    api 'com.amazonaws:aws-lambda-java-events:1.1.0'
    api 'com.amazonaws:aws-java-sdk-cognitoidp:+'
    api 'com.amazonaws:aws-java-sdk-sns:+'
    api 'com.amazonaws:aws-java-sdk-cognitosync:+'
    api group: 'com.google.code.gson', name: 'gson', version: '2.8.2'
    api group: 'com.jayway.jsonpath', name: 'json-path', version: '2.3.0'
    api 'com.auth0:java-jwt:3.3.0'
}
```

7.1.1.3 Context's build.gradle example: Offer

Example of a context's project gradle build script containing the task to update AWS Lambda function code.

```
import jp.classmethod.aws.gradle.lambda.AWSLambdaUpdateFunctionCodeTask;

group 'balsick-aws'
version '1.0-SNAPSHOT'

apply plugin: 'java'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencyManagement {
    imports {
        mavenBom 'com.amazonaws:aws-java-sdk-bom:1.11.228'
    }
}

dependencies {
    implementation project(':utils')
    testCompile group: 'junit', name: 'junit', version: '4.12'
}

task updateLambdaFunctions {
    updateLambdaFunctionsTask(project, ['UsedBookStoreThesis-ConfirmLockOffersFunction-
1B22MW81BYO1C',
                                        'UsedBookStoreThesis-CheckAndLockOffersFunction-1CDMUDUAKB6M5',
                                        'UsedBookStoreThesis-UnlockOffersFunction-191ZKKQXQ2K00',
                                        'UsedBookStoreThesis-GetOffersFunction-1X9JIMXPBAO9E',
                                        'UsedBookStoreThesis-CreateOfferFunction-PJPH3IGOTQ9L'])
}
```

7.1.2 Java code samples

7.1.2.1 ProxyRequest mapping

Java class of the Utils project mapping a Proxy Request coming from an API Gateway metho and passed as argument to a Lambda function. Other than mapping through the POJO part of the class, it contains some logics that, by using JsonPath, provide data contained deep in the request's body or in the context of the authorizer.

```
package it.balsick.cloud.aws.thesis.data;

import com.amazonaws.services.lambda.runtime.Context;
import com.fasterxml.jackson.annotation.JsonIgnore;
import com.jayway.jsonpath.JsonPath;
import com.jayway.jsonpath.ReadContext;

import java.util.Map;

public abstract class ProxyRequest<T extends Bean> implements Request<T> {

  private String resource;
  private String path;
  private String httpMethod;
  private Map<String, String> headers;
  private Map<String, String> queryStringParameters;
  private Map<String, String> pathParameters;
  private Map<String, Object> stageVariables;
  private Map<String, Object> requestContext;
  private Context context;
  private String body;
  private boolean isBase64Encoded;

  @JsonIgnore
  private transient ReadContext ctx;

  public ProxyRequest() {
  }

  public final String getResource() {
    return resource;
  }

  public final void setResource(String resource) {
    this.resource = resource;
  }

  public final String getPath() {
    return path;
  }

  public final void setPath(String path) {
    this.path = path;
```

```
}

public final String getHttpMethod() {
    return httpMethod;
}

public final void setHttpMethod(String httpMethod) {
    this.httpMethod = httpMethod;
}

public final Map<String, String> getHeaders() {
    return headers;
}

public final void setHeaders(Map<String, String> headers) {
    this.headers = headers;
}

public final Map<String, String> getQueryStringParameters() {
    return queryStringParameters;
}

public final void setQueryStringParameters(Map<String, String> queryStringParameters) {
    this.queryStringParameters = queryStringParameters;
}

public final Map<String, String> getPathParameters() {
    return pathParameters;
}

public final void setPathParameters(Map<String, String> pathParameters) {
    this.pathParameters = pathParameters;
}

public final Map<String, Object> getStageVariables() {
    return stageVariables;
}

public final void setStageVariables(Map<String, Object> stageVariables) {
    this.stageVariables = stageVariables;
}

public final Map<String, Object> getRequestContext() {
    return requestContext;
}

public final void setRequestContext(Map<String, Object> requestContext) {
    this.requestContext = requestContext;
}

public final String getBody() {
    return body;
}

public final void setBody(String body) {
    this.body = body;
}

public final boolean isBase64Encoded() {
```

```
    return isBase64Encoded;
}

public final void setBase64Encoded(boolean isBase64Encoded) {
    this.isBase64Encoded = isBase64Encoded;
}

public Context getContext() {
    return context;
}

public void setContext(Context context) {
    this.context = context;
}

/**
 * Get from the body or from the query string parameters the object having
 * the passed string as key in the body or in the query string parameters.
 *
 * @param key
 * @return
 */
public Object standardGet(String key) {
    try {
        if (ctx == null)
            ctx = JsonPath.parse(getBody());
        return ctx.read("$." + key);
    } catch (Exception ignored) {
    }
    try {
        return getQueryStringParameters().get(key);
    } catch (Exception ex) {
        return null;
    }
}

public Object getFromAuthorizerContext(String key) {
    return standardGet("context.authorizer."+key);
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof ProxyRequest)) return false;

    ProxyRequest<?> that = (ProxyRequest<?>) o;

    if (isBase64Encoded != that.isBase64Encoded) return false;
    if (resource != null ? !resource.equals(that.resource) : that.resource != null)
        return false;
    if (path != null ? !path.equals(that.path) : that.path != null) return false;
    if (httpMethod != null ? !httpMethod.equals(that.httpMethod) : that.httpMethod != null)
        return false;
    if (headers != null ? !headers.equals(that.headers) : that.headers != null) return false;
    if (queryStringParameters != null ?
            !queryStringParameters.equals(that.queryStringParameters) :
            that.queryStringParameters != null)
        return false;
    if (pathParameters != null ?
```

```
            !pathParameters.equals(that.pathParameters) :
            that.pathParameters != null)
          return false;
        if (stageVariables != null ?
            !stageVariables.equals(that.stageVariables) :
            that.stageVariables != null)
          return false;
        if (context != null ? !context.equals(that.context) : that.context != null)
          return false;
        return body != null ? body.equals(that.body) : that.body == null;

      }

      @Override
      public int hashCode() {
        int result = resource != null ? resource.hashCode() : 0;
        result = 31 * result + (path != null ? path.hashCode() : 0);
        result = 31 * result + (httpMethod != null ? httpMethod.hashCode() : 0);
        result = 31 * result + (headers != null ? headers.hashCode() : 0);
        result =
            31 * result + (queryStringParameters != null ?
                queryStringParameters.hashCode() :
                0);
        result = 31 * result + (pathParameters != null ? pathParameters.hashCode() : 0);
        result = 31 * result + (stageVariables != null ? stageVariables.hashCode() : 0);
        result = 31 * result + (context != null ? context.hashCode() : 0);
        result = 31 * result + (body != null ? body.hashCode() : 0);
        result = 31 * result + (isBase64Encoded ? 1 : 0);
        return result;
      }
    }
```

7.1.2.2 Interface with DynamoDB: DDBInterface

Java class containing convenience methods to access, both in reading and in writing, more easily to the DynamoDB database corresponding to the Java Bean mapping it.

These methods heavily use Java Reflection and so they might be suboptimal for AWS Lambda Function usage, though allowing easiest access to the developer.

```
package it.balsick.cloud.aws.thesis.data;

import com.amazonaws.regions.Regions;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBIndexHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBIndexRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBMapper;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBQueryExpression;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBRangeKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.TableWriteItems;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;

import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import it.balsick.cloud.aws.thesis.data.exception.ItemNotFoundException;
import it.balsick.cloud.aws.thesis.data.exception.TooManyResultsException;

public class DDBInterface {

  private DDBInterface() {
  }

  public static <T extends Bean> T getItem(Class<T> tClass, String hashKey)
      throws TooManyResultsException, ItemNotFoundException {
    return getItem(tClass, hashKey, null, null);
```

```
  }

  public static <T extends Bean> T getItem(Class<T> tClass, String hashKey, String rangeKey, String
index)
      throws TooManyResultsException, ItemNotFoundException {
    System.out.println("GetItem\t-\tclass:[" + (tClass == null ?
        "null" :
        tClass.getName()) + "] keys:[" + hashKey
        + ", " + rangeKey + "] index:[" + index + "]");
    List<T> items = getItems(tClass, hashKey, rangeKey, index);
    if (items != null && items.size() > 1)
      throw new TooManyResultsException();
    if (items == null || items.isEmpty())
      throw new ItemNotFoundException();
    return items.get(0);
  }

  private static boolean isAnnotationOnIndex(Annotation a, String index) {
    if (a == null)
      return false;
    String[] gsis = (String[]) getAnnotationValue(a, "globalSecondaryIndexNames");
    if (gsis == null || gsis.length == 0)
      gsis = new String[]{(String) getAnnotationValue(a, "globalSecondaryIndexName")};
    return Arrays.stream(gsis).anyMatch(s -> s.equals(index));
  }

  private static <T extends Bean> T buildBean(Class<T> tClass, String hashKey, String rangeKey, String
index)
      throws Exception {
    if (hashKey == null)
      throw new IllegalArgumentException();
    if (index == null)
      return (T) tClass.newInstance().with(new String[]{hashKey, rangeKey});
    T bean = tClass.newInstance();
    for (Method method : tClass.getDeclaredMethods()) {
      DynamoDBIndexHashKey a = method.getAnnotation(DynamoDBIndexHashKey.class);
      DynamoDBIndexRangeKey b = method.getAnnotation(DynamoDBIndexRangeKey.class);
      if (a == null && (b == null || rangeKey == null))
        continue;
      String value;
      if (!isAnnotationOnIndex(Utils.nvl(a, b), index))
        continue;
      value = hashKey;
      String fieldName = method.getName();
      if (fieldName.startsWith("get")) {
        fieldName = fieldName.replaceFirst("get", "");
        fieldName = (fieldName.charAt(0) + "").toLowerCase() + fieldName.substring(1);
      }
      Field field = tClass.getDeclaredField(fieldName);
      field.setAccessible(true);
      field.set(bean, value);
    }
    return bean;
  }

  private static Object getAnnotationValue(Annotation annotation, String methodName) {
    try {
      return annotation.getClass().getMethod(methodName).invoke(annotation);
    } catch (IllegalAccessException | IllegalArgumentException | InvocationTargetException |
```

X

```
NoSuchMethodException
            | SecurityException e) {
        e.printStackTrace();
        return null;
    }
  }

  public static <T extends Bean> List<T> getItems(Class<T> tClass, String hashKey, String rangeKey,
String index) {
    System.out.println("GetItems\t-\tclass:[" + (tClass == null ?
        "null" :
        tClass.getName()) + "] keys:[" + hashKey
        + ", " + rangeKey + "] index:[" + index + "]");

    T shape;
    try {
      shape = buildBean(tClass, hashKey, rangeKey, index);
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }

    KeyCondition kc = getKeyCondition(shape, index);
    System.out.println("KeyCondition: " + kc.keyConditionExpression);

    DynamoDBQueryExpression<T> queryExpression = new DynamoDBQueryExpression<T>()
        .withKeyConditionExpression(kc.keyConditionExpression)
        .withExpressionAttributeValues(kc.values);
    if (!kc.expressionAttributeNames.isEmpty())
      queryExpression =
          queryExpression.withExpressionAttributeNames(kc.expressionAttributeNames);
    if (index != null)
      queryExpression = queryExpression.withIndexName(index).withConsistentRead(false);
    System.out.println("QueryExpression created");

    DynamoDBMapper
        mapper =
        new DynamoDBMapper(
            AmazonDynamoDBClientBuilder.standard().withRegion(Regions.EU_WEST_1)
                .build());
    System.out.println("mapper created");
    return mapper.query(tClass, queryExpression);
  }

  public static <T extends Bean> List<T> getItems(Class<T> class1, List<String> hashKeys, String index)
{
    return hashKeys.parallelStream()
        .map(key -> getItems(class1, key, null, index))
        .flatMap(Collection::stream)
        .distinct()
        .collect(Collectors.toList());
  }

  public static <T extends Bean> List<T> getItems(Class<T> class1, List<String> hashKeys, List<String>
rangeKeys, String index) {
    if (hashKeys == null)
      throw new IllegalArgumentException("Hash keys null");
    if (rangeKeys == null)
      throw new IllegalArgumentException("Range keys null");
```

```
    if (hashKeys.size() != rangeKeys.size())
      throw new IllegalArgumentException("Hash keys and range keys of different size");
    return IntStream.range(0, hashKeys.size())
        .mapToObj(i -> getItems(class1, hashKeys.get(i), rangeKeys.get(i), index))
        .flatMap(Collection::stream)
        .distinct()
        .collect(Collectors.toList());
  }

  private static <T extends Bean, A extends Annotation> KeyMethod
getBeanKeyAttributeName(Class<T> tClass,
                                                  Class<A> annClass) {
    return getBeanKeyAttributeName(tClass, annClass, null);
  }

  private static <T extends Bean, A extends Annotation> KeyMethod
getBeanKeyAttributeName(Class<T> tClass,
                                                  Class<A> annClass, Predicate<A> predicate) {
    Field
        field =
        Arrays.stream(tClass.getDeclaredFields())
            .filter(f -> f.isAnnotationPresent(annClass))
            .findAny()
            .orElse(null);
    if (field != null) {
      String s = getAttributeName(field.getAnnotation(annClass));
      if (s != null)
        return new KeyMethod(s, null);
      return new KeyMethod(field.getName(), null);
    }
    Method
        method =
        Arrays.stream(tClass.getDeclaredMethods())
            .filter(f -> f.isAnnotationPresent(annClass))
            .filter(m -> predicate == null || predicate.test(m.getAnnotation(annClass)))
            .findAny()
            .orElse(null);
    if (method == null)
      throw new IllegalArgumentException();
    String key = method.getName();
    if (key.startsWith("get")) {
      key = key.replaceFirst("get", "");
      key = (key.charAt(0) + "").toLowerCase() + key.substring(1);
    }
    String s = getAttributeName(annClass, method);
    return new KeyMethod(s != null ? s : key, method);
  }

  private static <A extends Annotation> String getAttributeName(Class<A> annClass, Method method) {
    return getAttributeName(method.getAnnotation(annClass));
  }

  private static <A extends Annotation> String getAttributeName(A annotation) {
    try {
      String s = (String) getAnnotationValue(annotation, "attributeName");
      if (!(s != null && s.isEmpty()))
        return s;
    } catch (SecurityException | IllegalArgumentException | NullPointerException ignored) {
    }
```

```
        return null;
    }

    private static <T extends Bean> KeyCondition getKeyCondition(T shape, String secondaryIndexName)
{
        return Arrays.stream(shape.getClass().getMethods()).peek(f -> f.setAccessible(true))
                .filter(method -> secondaryIndexName != null
                        && (isAnnotationOnIndex(method.getAnnotation(DynamoDBIndexHashKey.class),
                        secondaryIndexName)
                        || isAnnotationOnIndex(method.getAnnotation(DynamoDBIndexRangeKey.class),
                        secondaryIndexName))
                        || secondaryIndexName == null && (method.isAnnotationPresent(
                        DynamoDBHashKey.class)
                        || method.isAnnotationPresent(DynamoDBRangeKey.class)))
                .map(method -> {
                    String key = method.getName();
                    if (key.startsWith("get")) {
                        key = key.replaceFirst("get", "");
                        key = (key.charAt(0) + "").toLowerCase() + key.substring(1);
                    }
                    String s = Utils.nvl(() -> getAttributeName(DynamoDBIndexHashKey.class, method),
                            () -> getAttributeName(DynamoDBIndexRangeKey.class, method),
                            () -> getAttributeName(DynamoDBHashKey.class, method),
                            () -> getAttributeName(DynamoDBRangeKey.class, method),
                            () -> getAttributeName(DynamoDBAttribute.class, method));
                    return new KeyMethod(Utils.nvl(s, key), method);
                }).map(current -> {
                    Map<String, String> expressionAttributeNames = new HashMap<>();
                    if (DynamoDBReservedWords.isReservedWord(current.key)) {
                        String k = String.format("#%s%d", current.key.substring(0, 3),
                                System.currentTimeMillis() % 1000);
                        expressionAttributeNames.put(k, current.key);
                        current.key = k;
                    }
                    String
                            k =
                            String.format("%s = :%s", current.key,
                                    Utils.nvl(expressionAttributeNames.get(current.key),
                                            current.key));
                    AttributeValue av = new AttributeValue();
                    Object value;
                    try {
                        value = current.field.invoke(shape);
                    } catch (IllegalArgumentException | IllegalAccessException | InvocationTargetException e) {
                        e.printStackTrace();
                        return null;
                    }
                    if (value == null)
                        return null;
                    if (current.field.getReturnType().isAssignableFrom(Boolean.class))
                        av.withBOOL((Boolean) value);
                    else if (current.field.getReturnType().isAssignableFrom(Number.class))
                        av.withN(((Number) value).toString());
                    else if (current.field.getReturnType().isInstance(""))
                        av.withS(value.toString());
                    Map<String, AttributeValue> map = new HashMap<>();
                    map.put(String.format(":%s",
                            Utils.nvl(expressionAttributeNames.get(current.key), current.key)), av);
                    return new KeyCondition(k, map, expressionAttributeNames);
```

```
        }).filter(Objects::nonNull).reduce((total, current) -> {
            Map<String, AttributeValue> map = new HashMap<>();
            map.putAll(total.values);
            map.putAll(current.values);
            Map<String, String> expressionAttributeNames = new HashMap<>();
            expressionAttributeNames.putAll(total.expressionAttributeNames);
            expressionAttributeNames.putAll(current.expressionAttributeNames);
            return new KeyCondition(
                    total.keyConditionExpression + " and " + current.keyConditionExpression,
                    map, expressionAttributeNames);
        }).orElse(null);
}

public static <T extends Bean, Y extends Request<T>> T updateItem(T src, Y request) {
    T dst = null;
    try {
        dst = (T) src.getClass().newInstance();
        dst = (T) dst.with(src.getKeys());
    } catch (NullPointerException | IllegalArgumentException ignored) {
    } catch (InstantiationException | IllegalAccessException e) {
        e.printStackTrace();
        return null;
    }
    Map<String, Field> requestMap = mapFields(request);

    Map<String, Field> sourceMap = mapFields(src);
    T dest = dst;
    requestMap.forEach((key, value) -> {
        try {
            value.setAccessible(true);
            sourceMap.get(key).setAccessible(true);
            sourceMap.get(key).set(dest, Utils.nvl(requestMap.get(key).get(request),
                    sourceMap.get(key).get(src)));
        } catch (IllegalArgumentException | IllegalAccessException e) {
            e.printStackTrace();
        } catch (NullPointerException ignored) {
        }
    });

    return updateItem(dst);
}

public static <T extends Bean> T updateItem(T src) {
    new DynamoDBMapper(
            AmazonDynamoDBClientBuilder.standard().withRegion("eu-west-1").build()).save(src);
    return src;
}

private static <T extends Bean> Class<T> getTClass(Request<T> request) {
    Type[] genericTypes;
    if (request.getClass().getGenericSuperclass() instanceof ParameterizedType)
        genericTypes =
                ((ParameterizedType) request.getClass()
                        .getGenericSuperclass()).getActualTypeArguments();
    else {
        genericTypes =
                ((ParameterizedType) request.getClass()
                        .getGenericInterfaces()[0]).getActualTypeArguments();
    }
```

```
    return (Class<T>) genericTypes[0];
  }

  public static <T extends Bean> T postItem(Request<T> request) {
    try {
      return updateItem(getTClass(request).newInstance(), request);
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }

  private static Map<String, Field> mapFields(Object object) {
    return Arrays.stream(object.getClass().getDeclaredFields())
        .filter(field -> !field.isAnnotationPresent(BeanCopyIgnore.class))
        .collect(Collectors.toMap(Field::getName, Function.identity()));
  }

  private static <T extends Bean> String getBeanKeyAttributeName(Class<T> t) {
    return getBeanKeyMethodAttribute(t).key;
  }

  private static <T extends Bean> KeyMethod getBeanKeyMethodAttribute(Class<T> t) {
    return getBeanKeyAttributeName(t, DynamoDBHashKey.class);
  }

  public static <T extends Bean> void delete(Class<? extends T> clazz, List<T> items) {
    List<TableWriteItems> itemsToWrite = new ArrayList<>();
    List<Object> hashAndRangeKeys = items.stream()
        .filter(c -> c.getKeys().length == 2 && c.getKeys()[0] != null && c.getKeys()[1] != null)
        .map(T::getKeys)
        .flatMap(Arrays::stream)
        .filter(Objects::nonNull)
        .collect(Collectors.toList());
    List<Object> hashOnlyKeys = items.stream()
        .filter(c -> c.getKeys().length == 1 || c.getKeys().length == 2 && Utils.isEmpty(
            c.getKeys()[1]))
        .map(T::getKeys)
        .flatMap(Arrays::stream)
        .filter(Objects::nonNull)
        .distinct()
        .collect(Collectors.toList());
    if (!hashAndRangeKeys.isEmpty())
      itemsToWrite.add(
          new TableWriteItems(clazz.getAnnotation(DynamoDBTable.class).tableName())
              .withHashAndRangeKeysToDelete(getBeanKeyAttributeName(clazz),
                  getBeanKeyAttributeName(clazz, DynamoDBRangeKey.class).key,
                  hashAndRangeKeys.toArray()));
    if (!hashOnlyKeys.isEmpty())
      itemsToWrite.add(
          new TableWriteItems(clazz.getAnnotation(DynamoDBTable.class).tableName())
              .withHashOnlyKeysToDelete(getBeanKeyAttributeName(clazz),
                  hashOnlyKeys.toArray()));
    System.out.println("Trying to delete "+clazz.getName()+"\nHashOnlyKeys:");
    hashOnlyKeys.forEach(System.out::println);
    System.out.println("HashAndRangeKeys:");
    hashAndRangeKeys.forEach(System.out::println);
    AmazonDynamoDB
        client =
```

```
            AmazonDynamoDBClientBuilder.standard().withRegion("eu-west-1").build();
        DynamoDB dynamoDB = new DynamoDB(client);
        dynamoDB.batchWriteItem(itemsToWrite.toArray(new TableWriteItems[itemsToWrite.size()]));
    }

    public static <T extends Bean> void delete(Class<T> tClass, String hashKey, String rangeKey, String
index)
            throws ItemNotFoundException {
        List<T> items = getItems(tClass, hashKey, rangeKey, index);
        if (items == null || items.isEmpty())
            throw new ItemNotFoundException();
        Utils.splitIntoChunks(items, 25).forEach(list -> delete(tClass, list));
    }

    public static <T extends Bean> void delete(T item) {
        delete(item.getClass(), Collections.singletonList(item));
    }

    public static <T extends Bean> List<T> getItems(Class<T> cls) {
        return getItems(cls, (String) null, (String) null, (String) null);
    }

    private static class KeyMethod {
        String key;
        Method field;

        KeyMethod(String key, Method field) {
            this.key = key;
            this.field = field;
        }

    }

    private static final class KeyCondition {
        final String keyConditionExpression;
        final Map<String, AttributeValue> values;
        final Map<String, String> expressionAttributeNames;

        private KeyCondition(String kce, Map<String, AttributeValue> v, Map<String, String>
expressionAttributeNames) {
            this.keyConditionExpression = kce;
            this.values = v;
            this.expressionAttributeNames = expressionAttributeNames;
        }
    }

}
```

7.1.2.3 Result

Abstract class representing a lambda function result.

```
package it.balsick.cloud.aws.thesis.data;

public abstract class Result {

  final int statusCode;

  protected Result(int result) {
    this.statusCode = result;
  }

  public int getStatusCode() {
    return statusCode;
  }

}
```

7.1.2.4 SuccessResult

Abstract parametric class representing a successful lambda function result.

```
package it.balsick.cloud.aws.thesis.data;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.google.gson.Gson;

import java.util.Collections;
import java.util.Map;

public abstract class SuccessResult<T> extends Result {

  @JsonIgnore
  public transient final T data;

  public SuccessResult(T data) {
    super(200);
    this.data = data;
  }

  public String getBody() {
    return new Gson().toJson(data);
  }

  public final static class EmptySuccessResult extends SuccessResult<Map<Void, Void>> {
    public EmptySuccessResult() {
      super(Collections.emptyMap());
    }
  }

}
```

7.1.2.5 CreateBook

Example of a Lambda Function handler in Java. The *handler* method is the entrypoint of the lambda function. It receives as input an object that maps the function's input, a Context object containing general information on the running lambda function, and it provides a result that needs to be compliant to the Proxy result (cfr. 7.1.2.3) for Proxy API Gateway method calls, or it is free otherwise.

```
package it.balsick.cloud.aws.thesis.book;

import com.amazonaws.services.lambda.runtime.Context;
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.List;
import java.util.Map;

import javax.net.ssl.HttpsURLConnection;

import it.balsick.cloud.aws.thesis.book.request.BookRequest;
import it.balsick.cloud.aws.thesis.data.DDBInterface;
import it.balsick.cloud.aws.thesis.data.ErrorResult;
import it.balsick.cloud.aws.thesis.data.ErrorType;
import it.balsick.cloud.aws.thesis.data.Result;

public class CreateBook {

  private final static String
      GOOGLE_API_BASE =
      "https://www.googleapis.com/books/v1/volumes?q=isbn:";

  public Result handler(BookRequest request, Context context) {
    System.out.println(
        "Received it.balsick.cloud.aws.thesis.book request\t-\tisbn:[" + request.getIsbn() + "]
author:[" + request
            .getAuthor() + "]");
    String _url = GOOGLE_API_BASE + request.getIsbn()
        + "&fields=items(volumeInfo(title,authors,publishedDate,pageCount,language))";
    try {
      URL url = new URL(_url);
      HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
      conn.setRequestMethod("GET");

      if (conn.getResponseCode() != 200)
        return new ErrorResult(ErrorType.NOT_FOUND);
      BufferedReader
          reader =
          new BufferedReader(new InputStreamReader(conn.getInputStream()));
      String line;
      String result = "";
```

```
            while ((line = reader.readLine()) != null)
                result += line + "\n";
            reader.close();
            Gson gson = new Gson();
            Map<String, Object> map = gson.fromJson(result, new TypeToken<Map<String, Object>>() {
            }.getType());
            List<Object> items = (List<Object>) map.get("items");
            if (items == null || items.isEmpty())
                return new ErrorResult(ErrorType.NOT_FOUND);
            Map<String, Object> item = (Map<String, Object>) items.get(0);
            Map<String, Object> itemInfo = (Map<String, Object>) item.get("volumeInfo");
            Book book = new Book();
            book.setIsbn(request.getIsbn());
            book.setAuthor(((List<String>) itemInfo.get("authors")).get(0));
            book.setTitle((String) itemInfo.get("title"));
            Number pageCount = (Number) itemInfo.get("pageCount");
            if (pageCount != null)
                book.setPageCount(pageCount.intValue());
            String publishedDate = (String) itemInfo.get("publishedDate");
            if (publishedDate != null)
                book.setPublishedDate(new Integer(publishedDate));
            book.setLanguage((String) itemInfo.get("language"));
            DDBInterface.updateItem(book);
        } catch (Exception ex) {
            System.err.println("Error querying: "+_url);
            ex.printStackTrace();
            return new ErrorResult(ErrorType.BAD_REQUEST);
        }
        return new GetBook().handler(request, context);
    }

}
```

7.1.3 Node.js code samples

7.1.3.1 Courier Sign Up

Example of a Lambda function handler in Node.js. The handler method is the entrypoint of the lambda function. It receives an event object as input mapping the lambda function's input and returns an object that, if the lambda function is called by a Proxy API Gateway method, needs to be compliant to the proxy response.

```
const utils = require('../utils');

const crypto = require('crypto');

exports.handler = async event => {
  let alreadyExists = false;
  try {
    let res = await utils.dynamodb.getItem({
      Key: utils.valueToItem({
        courier: event.courier
      }),
      TableName: "CourierRegistrations"
    }).promise();
    if (res.Item)
      alreadyExists = true;
  } catch (e) {
  }
  if (alreadyExists) {
    console.log("Item already exists with courier = " + event.courier);
    return {statusCode: 403};
  }
  const salt = utils.randomString(8, 16);
  const hmac = crypto.createHmac('sha256', salt);
  hmac.update(event.password);
  const hashedPassword = hmac.digest('hex');
  await utils.dynamodb.putItem({
    Item: utils.valueToItem({
      courier: event.courier,
      salt: salt,
      hashedPassword: hashedPassword
    }),
    TableName: "CourierRegistrations"
  }).promise();
  let message = {
    courier: event.courier
  };
  let snsMessageParams = {
    TopicArn: "arn:aws:sns:eu-west-1:468165443921:UsedBookStoreThesis-
NewCourierRegistered",
    Message: JSON.stringify(message)
  };
  await utils.sns.publish(snsMessageParams).promise();
  return {statusCode: 200};
};
```

XX

7.1.3.2 Notify all couriers for assignment

```
const utils = require('../utils');

const sendEmailParamsTemplate = {
  Destination: {
    BccAddresses: [
      'aws-thesis-assignments@balsick.it'
    ],
  },
  Message: {
    Body: {
      Html: {
        Charset: "UTF-8",
      },
      Text: {
        Charset: "UTF-8",
      }
    },
    Subject: {
      Charset: 'UTF-8',
    }
  },
  Source: 'aws-thesis-assignments@balsick.it'
};

const htmlData = (dims, parcelInfo) => `<html>There is a parcel yet to been assigned!<br/>
            If you are interested in delivering it, assign it to yourself.<br/>
            Here are some infos:<br/>
            weight: ${parcelInfo.size.weight}<br/>
            dimensions: ${dims[0]}x${dims[1]}x${dims[2]}</html>`;

const textData = (dims, parcelInfo) => `There is a parcel yet to been assigned, identified as
${parcelInfo.parcel}!\n
            If you are interested in delivering it, assign it to yourself.\n
            Here are some infos:\n\t
            weight:${parcelInfo.size.weight}\n\t
            dimensions:${dims[0]}x${dims[1]}x${dims[2]}`;

const subjectData = parcelInfo => `New assignment on parcel [${parcelInfo.parcel}]`;

module.exports.handler = async event => {

  let parcelInfoResult = await utils.dynamodb.getItem({
    TableName: "ParcelInfo",
    Key: {
      parcel: {
        S: event.parcel
      }
    }
  }).promise();
  let parcelInfo = utils.ItemToValue(parcelInfoResult.Item);

  let dims = parcelInfo.size.dimensions;
  dims = dims.sort((d1, d2) => d2 - d1);

  const getCouriersParams = {
    TableName: "Couriers",
```

```
    ExpressionAttributeNames: {EM: "email", VN: "visualName"},
    ProjectionExpression: "#EM, #VN",
    Limit: 50,
  };

  const sendResults = [];
  let courierResult;
  do {
    courierResult = await utils.dynamodb.scan(getCouriersParams).promise();
    let sendEmailParams = Object.assign({}, sendEmailParamsTemplate);
    sendEmailParams.Destination.ToAddresses = courierResult.Items
      .map(utils.ItemToValue)
      .map(courier => courier.email || courier.EM);
    sendEmailParams.Message.Body.Html.Data = htmlData(dims, parcelInfo);
    sendEmailParams.Message.Body.Text.Data = textData(dims, parcelInfo);
    sendEmailParams.Message.Subject.Data = subjectData(parcelInfo);
    let sendResult = await utils.ses.sendEmail(sendEmailParams).promise();
    sendResults.push(sendResult.MessageId);
  }
  while (courierResult.LastEvaluatedKey /*&& courierResult.Count != 50 */);

  return sendResults;
};
```

7.1.4 Cart API CloudFormation


```
{
 "AWSTemplateFormatVersion": "2010-09-09", "Parameters": {
 "DomainNameParameter": {
  "Type": "String", "Description": "The unique name of the domain"
 }
}, "Resources": {
 "Api": {
  "Type": "AWS::ApiGateway::RestApi", "Properties": {
   "Name": {
    "Fn::Join": [
     "-", [
      "CartApi", {
       "Ref": "DomainNameParameter"
      }
     ]
    ]
   }
  }
 }, "ApiGatewayCloudWatchLogsRole": {
  "Type": "AWS::IAM::Role", "Properties": {
   "AssumeRolePolicyDocument": {
    "Version": "2012-10-17", "Statement": [
     {
      "Effect": "Allow", "Principal": {
       "Service": [
        "apigateway.amazonaws.com"
       ]
      }, "Action": [
       "sts:AssumeRole"
      ]
     }
    ]
   }, "Policies": [
    {
     "PolicyName": "ApiGatewayLogsPolicy", "PolicyDocument": {
     "Version": "2012-10-17", "Statement": [
      {
       "Effect": "Allow", "Action": [
        "logs:CreateLogGroup", "logs:CreateLogStream", "logs:DescribeLogGroups",
"logs:DescribeLogStreams",
        "logs:PutLogEvents", "logs:GetLogEvents", "logs:FilterLogEvents"
       ], "Resource": "*"
      }
     ]
    }
    }
   ]
  }
 }, "CartLambdaFunctionsRole": {
  "Type": "AWS::IAM::Role", "Properties": {
   "AssumeRolePolicyDocument": {
    "Version": "2012-10-17", "Statement": [
     {
      "Effect": "Allow", "Principal": {
       "Service": [
        "lambda.amazonaws.com"
       ]
```

```
        }, "Action": [
          "sts:AssumeRole"
        ]
        }
      ]
    }, "Path": "/", "Policies": [
      {
        "PolicyName": "root", "PolicyDocument": {
        "Version": "2012-10-17", "Statement": [
          {
            "Effect": "Allow", "Action": [
            "logs:*"
          ], "Resource": "arn:aws:logs:*:*:*"
          }, {
            "Effect": "Allow", "Action": [
              "dynamodb:GetItem", "dynamodb:GetRecords", "dynamodb:PutItem", "dynamodb:Query",
"dynamodb:UpdateItem"
            ], "Resource": [
              {
                "Ref": "Carts"
              }, {
                "Fn::Join": [
                  "", [
                    {
                      "Ref": "Carts"
                    }, "/*"
                  ]
                ]
              }
            ]
          }, {
            "Effect": "Allow", "Action": [
              "dynamodb:GetItem", "dynamodb:GetRecords", "dynamodb:Query"
            ], "Resource": [
              {
                "Ref": "DomainSettings"
              }, {
                "Fn::Join": [
                  "", [
                    {
                      "Ref": "DomainSettings"
                    }, "/*"
                  ]
                ]
              }, {
                "Ref": "ProductDomainSettings"
              }, {
                "Fn::Join": [
                  "", [
                    {
                      "Ref": "ProductDomainSettings"
                    }, "/*"
                  ]
                ]
              }
            ]
          }
        ]
      }
    }
```

```
      }
     ]
    }
   }, "PrivateCartLambdaFunctionRole": {
    "Type": "AWS::IAM::Role", "Properties": {
     "AssumeRolePolicyDocument": {
      "Version": "2012-10-17", "Statement": [
       {
        "Effect": "Allow", "Principal": {
        "Service": [
         "lambda.amazonaws.com"
        ]
       }, "Action": [
        "sts:AssumeRole"
       ]
       }
      ]
     }, "Path": "/", "Policies": [
      {
       "PolicyName": "root", "PolicyDocument": {
       "Version": "2012-10-17", "Statement": [
        {
         "Effect": "Allow", "Action": [
         "logs:*"
        ], "Resource": "arn:aws:logs:*:*:*"
        }, {
         "Effect": "Allow", "Action": [
          "dynamodb:GetItem", "dynamodb:GetRecords", "dynamodb:PutItem", "dynamodb:Query",
"dynamodb:UpdateItem"
         ], "Resource": [
          {
           "Ref": "DomainSettings"
          }, {
           "Fn::Join": [
            "", [
             {
              "Ref": "DomainSettings"
             }, "/*"
            ]
           ]
          }, {
           "Ref": "ProductDomainSettings"
          }, {
           "Fn::Join": [
            "", [
             {
              "Ref": "ProductDomainSettings"
             }, "/*"
            ]
           ]
          }
         ]
        }
       ]
      }
      }
     ]
    }
   }, "AuthorizerLambdaFunctionRole": {
```

```
  "Type": "AWS::IAM::Role", "Properties": {
   "AssumeRolePolicyDocument": {
    "Version": "2012-10-17", "Statement": [
     {
      "Effect": "Allow", "Principal": {
      "Service": [
       "lambda.amazonaws.com"
      ]
     }, "Action": [
      "sts:AssumeRole"
     ]
     }
    ]
   }, "Path": "/", "Policies": [
    {
     "PolicyName": "root", "PolicyDocument": {
     "Version": "2012-10-17", "Statement": [
      {
       "Effect": "Allow", "Action": [
       "logs:*"
      ], "Resource": "arn:aws:logs:*:*:*"
      }, {
       "Effect": "Allow", "Action": [
        "dynamodb:GetItem", "dynamodb:GetRecords", "dynamodb:PutItem", "dynamodb:Query",
"dynamodb:UpdateItem"
       ], "Resource": "arn:aws:dynamodb:*:*:table/Registrations"
      }
     ]
    }
    }
   ]
  }
 }, "ApiStage": {
  "DependsOn": [
   "ApiGatewayAccount"
  ], "Type": "AWS::ApiGateway::Stage", "Properties": {
   "DeploymentId": {
    "Ref": "ApiDeployment"
   }, "MethodSettings": [
    {
     "DataTraceEnabled": true, "HttpMethod": "*", "LoggingLevel": "INFO", "ResourcePath": "/*"
    }
   ], "RestApiId": {
    "Ref": "Api"
   }, "StageName": "LATEST"
  }
 }, "ApiGatewayAccount": {
  "Type": "AWS::ApiGateway::Account", "Properties": {
   "CloudWatchRoleArn": {
    "Fn::GetAtt": [
     "ApiGatewayCloudWatchLogsRole", "Arn"
    ]
   }
  }
 }, "ApiDeployment": {
  "Type": "AWS::ApiGateway::Deployment", "Properties": {
   "RestApiId": {
    "Ref": "Api"
   }, "StageName": "DummyStage"
```

```
 }
},"CartsResource": {
 "Type": "AWS::ApiGateway::Resource", "Properties": {
  "RestApiId": {
   "Ref": "Api"
  }, "ParentId": {
   "Fn::GetAtt": [
    "Api", "RootResourceId"
   ]
  }, "PathPart": "carts"
 }
},"DomainSettingsResource": {
 "Type": "AWS::ApiGateway::Resource", "Properties": {
  "RestApiId": {
   "Ref": "Api"
  }, "ParentId": {
   "Fn::GetAtt": [
    "Api", "RootResourceId"
   ]
  }, "PathPart": "domain-settings"
 }
},"RegistrationsResource": {
 "Type": "AWS::ApiGateway::Resource", "Properties": {
  "RestApiId": {
   "Ref": "Api"
  }, "ParentId": {
   "Fn::GetAtt": [
    "Api", "RootResourceId"
   ]
  }, "PathPart": "registration"
 }
},"GetCartPermission": {
 "Type": "AWS::Lambda::Permission", "Properties": {
  "Action": "lambda:invokeFunction", "FunctionName": {
   "Ref": "GetCartFunction"
  }, "Principal": "apigateway.amazonaws.com", "SourceArn": {
   "Fn::Join": [
    "", [
     "arn:aws:execute-api:", {
      "Ref": "AWS::Region"
     }, ":", {
      "Ref": "AWS::AccountId"
     }, ":", {
      "Ref": "Api"
     }, "/LATEST/GET/carts"
    ]
   ]
  }
 }
},"AddToCartPermission": {
 "Type": "AWS::Lambda::Permission", "Properties": {
  "Action": "lambda:invokeFunction", "FunctionName": {
   "Ref": "AddToCartFunction"
  }, "Principal": "apigateway.amazonaws.com", "SourceArn": {
   "Fn::Join": [
    "", [
     "arn:aws:execute-api:", {
      "Ref": "AWS::Region"
     }, ":", {
```

```
          "Ref": "AWS::AccountId"
        }, ":", {
          "Ref": "Api"
        }, "/LATEST/PUT/carts"
      ]
    ]
   }
  }
 }, "ConfirmCartPermission": {
  "Type": "AWS::Lambda::Permission", "Properties": {
   "Action": "lambda:invokeFunction", "FunctionName": {
    "Ref": "ConfirmCartFunction"
   }, "Principal": "apigateway.amazonaws.com", "SourceArn": {
    "Fn::Join": [
     "", [
      "arn:aws:execute-api:", {
       "Ref": "AWS::Region"
      }, ":", {
       "Ref": "AWS::AccountId"
      }, ":", {
       "Ref": "Api"
      }, "/LATEST/POST/carts"
     ]
    ]
   }
  }
 }, "RemoveFromCartPermission": {
  "Type": "AWS::Lambda::Permission", "Properties": {
   "Action": "lambda:invokeFunction", "FunctionName": {
    "Ref": "RemoveFromCartFunction"
   }, "Principal": "apigateway.amazonaws.com", "SourceArn": {
    "Fn::Join": [
     "", [
      "arn:aws:execute-api:", {
       "Ref": "AWS::Region"
      }, ":", {
       "Ref": "AWS::AccountId"
      }, ":", {
       "Ref": "Api"
      }, "/LATEST/DELETE/carts"
     ]
    ]
   }
  }
 }, "LogInPermission": {
  "Type": "AWS::Lambda::Permission", "Properties": {
   "Action": "lambda:invokeFunction", "FunctionName": {
    "Ref": "LogInFunction"
   }, "Principal": "apigateway.amazonaws.com"
  }
 }, "SignUpPermission": {
  "Type": "AWS::Lambda::Permission", "Properties": {
   "Action": "lambda:invokeFunction", "FunctionName": {
    "Ref": "SignUpFunction"
   }, "Principal": "apigateway.amazonaws.com"
  }
 }, "UpdateCartDomainPermission": {
  "Type": "AWS::Lambda::Permission", "Properties": {
   "Action": "lambda:invokeFunction", "FunctionName": {
```

```
      "Ref": "UpdateCartDomainSettingsFunction"
    }, "Principal": "apigateway.amazonaws.com"
   }
  }, "UpdateCartDomainProductSettingsPermission": {
   "Type": "AWS::Lambda::Permission", "Properties": {
    "Action": "lambda:invokeFunction", "FunctionName": {
     "Ref": "UpdateCartDomainProductSettingsFunction"
    }, "Principal": "apigateway.amazonaws.com"
   }
  }, "GetCartRequest": {
   "DependsOn": "GetCartPermission", "Type": "AWS::ApiGateway::Method", "Properties": {
    "AuthorizationType": "CUSTOM", "AuthorizerId": {
     "Ref": "Authorizer"
    }, "HttpMethod": "GET", "Integration": {
     "Type": "AWS", "IntegrationHttpMethod": "POST", "Uri": {
      "Fn::Join": [
       "", [
        "arn:aws:apigateway:", {
         "Ref": "AWS::Region"
        }, ":lambda:path/2015-03-31/functions/", {
         "Fn::GetAtt": [
          "GetCartFunction", "Arn"
         ]
        }, "/invocations"
       ]
      ]
     }, "IntegrationResponses": [
      {
       "StatusCode": 200
      }, {
       "StatusCode": 401
      }
     ]
    }, "ResourceId": {
     "Ref": "CartsResource"
    }, "RestApiId": {
     "Ref": "Api"
    }, "MethodResponses": [
     {
      "StatusCode": 200
     }, {
      "StatusCode": 401
     }
    ]
   }
  }, "AddToCartRequest": {
   "DependsOn": "AddToCartPermission", "Type": "AWS::ApiGateway::Method", "Properties": {
    "AuthorizationType": "CUSTOM", "AuthorizerId": {
     "Ref": "Authorizer"
    }, "HttpMethod": "PUT", "Integration": {
     "Type": "AWS", "IntegrationHttpMethod": "POST", "Uri": {
      "Fn::Join": [
       "", [
        "arn:aws:apigateway:", {
         "Ref": "AWS::Region"
        }, ":lambda:path/2015-03-31/functions/", {
         "Fn::GetAtt": [
          "AddToCartFunction", "Arn"
         ]
```

```
        }, "/invocations"
       ]
      ]
     }, "IntegrationResponses": [
      {
       "StatusCode": 200
      }, {
       "StatusCode": 401
      }
     ], "RequestTemplates": {
      "application/json": {
       "Fn::Join": [
        "", [
         "{", "\"product\": \"$input.params('product')\"", "}"
        ]
       ]
      }
     }
    }, "RequestParameters": {
     "method.request.querystring.product": true
    }, "ResourceId": {
     "Ref": "CartsResource"
    }, "RestApiId": {
     "Ref": "Api"
    }, "MethodResponses": [
     {
      "StatusCode": 200
     }, {
      "StatusCode": 401
     }
    ]
   }
  }, "ConfirmCartRequest": {
   "DependsOn": "ConfirmCartPermission", "Type": "AWS::ApiGateway::Method", "Properties": {
    "AuthorizationType": "CUSTOM", "AuthorizerId": {
     "Ref": "Authorizer"
    }, "HttpMethod": "POST", "Integration": {
     "Type": "AWS", "IntegrationHttpMethod": "POST", "Uri": {
      "Fn::Join": [
       "", [
        "arn:aws:apigateway:", {
         "Ref": "AWS::Region"
        }, ":lambda:path/2015-03-31/functions/", {
         "Fn::GetAtt": [
          "ConfirmCartFunction", "Arn"
         ]
        }, "/invocations"
       ]
      ]
     }, "IntegrationResponses": [
      {
       "StatusCode": 200
      }, {
       "StatusCode": 401
      }
     ]
    }, "ResourceId": {
     "Ref": "CartsResource"
    }, "RestApiId": {
```

XXX

```
      "Ref": "Api"
    }, "MethodResponses": [
     {
      "StatusCode": 200
     }, {
      "StatusCode": 401
     }
    ]
   }
  }, "RemoveFromCartRequest": {
   "DependsOn": "RemoveFromCartPermission", "Type": "AWS::ApiGateway::Method", "Properties":
{
    "AuthorizationType": "CUSTOM", "AuthorizerId": {
     "Ref": "Authorizer"
    }, "HttpMethod": "POST", "Integration": {
     "Type": "AWS", "IntegrationHttpMethod": "POST", "Uri": {
      "Fn::Join": [
       "", [
        "arn:aws:apigateway:", {
         "Ref": "AWS::Region"
        }, ":lambda:path/2015-03-31/functions/", {
         "Fn::GetAtt": [
          "RemoveFromCartFunction", "Arn"
         ]
        }, "/invocations"
       ]
      ]
     }, "IntegrationResponses": [
      {
       "StatusCode": 200
      }, {
       "StatusCode": 401
      }
     ]
    }, "ResourceId": {
     "Ref": "CartsResource"
    }, "RestApiId": {
     "Ref": "Api"
    }, "MethodResponses": [
     {
      "StatusCode": 200
     }, {
      "StatusCode": 401
     }
    ]
   }
  }, "Authorizer": {
   "Type": "AWS::ApiGateway::Authorizer", "Properties": {
    "AuthorizerCredentials": {
     "Fn::GetAtt": [
      "AuthorizerLambdaFunctionRole", "Arn"
     ]
    }, "AuthorizerResultTtlInSeconds": "300", "AuthorizerUri": {
     "Fn::Join": [
      "", [
       "arn:aws:apigateway:", {
        "Ref": "AWS::Region"
       }, ":lambda:path/2015-03-31/functions/", {
        "Fn::GetAtt": [
```

```
            "CheckTokenFunction", "Arn"
           ]
          }, "/invocations"
         ]
        ]
      }, "Type": "TOKEN", "IdentitySource": "method.request.header.CartAuthToken", "Name":
"DefaultAuthorizer",
      "RestApiId": {
       "Ref": "Api"
      }
     }
    }, "SignUpRequest": {
     "DependsOn": "SignUpPermission", "Type": "AWS::ApiGateway::Method", "Properties": {
      "HttpMethod": "POST", "Integration": {
       "Type": "AWS", "IntegrationHttpMethod": "POST", "Uri": {
        "Fn::Join": [
         "", [
          "arn:aws:apigateway:", {
           "Ref": "AWS::Region"
          }, ":lambda:path/2015-03-31/functions/", {
           "Fn::GetAtt": [
            "SignUpFunction", "Arn"
           ]
          }, "/invocations"
         ]
        ]
       }, "IntegrationResponses": [
        {
         "StatusCode": 200
        }, {
         "StatusCode": 401
        }
       ], "RequestTemplates": {
        "application/json": {
         "Fn::Join": [
          "", [
           "{", "\"username\": \"$input.params('username')\",", "\"password\":
\"$input.params('password')\"", "}"
          ]
         ]
        }
       }
      }, "RequestParameters": {
       "method.request.querystring.username": true, "method.request.querystring.password": true
      }, "ResourceId": {
       "Ref": "RegistrationsResource"
      }, "RestApiId": {
       "Ref": "Api"
      }, "MethodResponses": [
       {
        "StatusCode": 200
       }, {
        "StatusCode": 401
       }
      ]
     }
    }, "LogInRequest": {
     "DependsOn": "LogInPermission", "Type": "AWS::ApiGateway::Method", "Properties": {
      "HttpMethod": "POST", "Integration": {
```

```
      "Type": "AWS", "IntegrationHttpMethod": "POST", "Uri": {
       "Fn::Join": [
        "", [
         "arn:aws:apigateway:", {
          "Ref": "AWS::Region"
         }, ":lambda:path/2015-03-31/functions/", {
          "Fn::GetAtt": [
           "LogInFunction", "Arn"
          ]
         }, "/invocations"
        ]
       ]
      }, "IntegrationResponses": [
       {
        "StatusCode": 200
       }, {
        "StatusCode": 401
       }
      ], "RequestTemplates": {
       "application/json": {
        "Fn::Join": [
         "", [
          "{", "\"username\": \"$input.params('username')\",", "\"password\":
\"$input.params('password')\"", "}"
         ]
        ]
       }
      }
     }, "RequestParameters": {
      "method.request.querystring.username": true, "method.request.querystring.password": true
     }, "ResourceId": {
      "Ref": "RegistrationsResource"
     }, "RestApiId": {
      "Ref": "Api"
     }, "MethodResponses": [
      {
       "StatusCode": 200
      }, {
       "StatusCode": 401
      }
     ]
    }
   }, "UpdateCartDomainProductSettingsRequest": {
    "DependsOn": [
     "UpdateCartDomainProductSettingsPermission"
    ], "Type": "AWS::ApiGateway::Method", "Properties": {
     "HttpMethod": "PUT", "Integration": {
      "Type": "AWS_PROXY", "IntegrationHttpMethod": "POST", "Uri": {
       "Fn::Join": [
        "", [
         "arn:aws:apigateway:", {
          "Ref": "AWS::Region"
         }, ":lambda:path/2015-03-31/functions/", {
          "Fn::GetAtt": [
           "UpdateCartDomainProductSettingsFunction", "Arn"
          ]
         }, "/invocations"
        ]
       ]
```

```
    }, "IntegrationResponses": [
     {
      "StatusCode": 200
     }, {
      "StatusCode": 401
     }
    ]
   }, "ResourceId": {
    "Ref": "DomainSettingsResource"
   }, "RestApiId": {
    "Ref": "Api"
   }, "MethodResponses": [
    {
     "StatusCode": 200
    }, {
     "StatusCode": 401
    }
   ]
  }
 }, "UpdateCartDomainSettingsRequest": {
  "DependsOn": [
   "UpdateCartDomainPermission"
  ], "Type": "AWS::ApiGateway::Method", "Properties": {
   "HttpMethod": "PATCH", "Integration": {
    "Type": "AWS_PROXY", "IntegrationHttpMethod": "POST", "Uri": {
     "Fn::Join": [
      "", [
       "arn:aws:apigateway:", {
        "Ref": "AWS::Region"
       }, ":lambda:path/2015-03-31/functions/", {
        "Fn::GetAtt": [
         "UpdateCartDomainSettingsFunction", "Arn"
        ]
       }, "/invocations"
      ]
     ]
    }, "IntegrationResponses": [
     {
      "StatusCode": 200
     }, {
      "StatusCode": 401
     }
    ]
   }, "ResourceId": {
    "Ref": "DomainSettingsResource"
   }, "RestApiId": {
    "Ref": "Api"
   }, "MethodResponses": [
    {
     "StatusCode": 200
    }, {
     "StatusCode": 401
    }
   ]
  }
 }, "GetCartFunction": {
  "Type": "AWS::Lambda::Function", "Properties": {
   "Handler": "it.balsick.cloud.aws.thesis.cart.GetCart::handler", "Code": {
    "S3Bucket": "lambdafunctions", "S3Key": "javafunctions.zip"
```

```
    }, "Runtime": "java8", "Role": {
      "Fn::GetAtt": [
        "CartLambdaFunctionsRole", "Arn"
      ]
    }
  }, "DependsOn": [
    "Carts"
  ]
}, "AddToCartFunction": {
  "Type": "AWS::Lambda::Function", "Properties": {
    "Handler": "it.balsick.cloud.aws.thesis.cart.AddToCart::handler", "Code": {
      "S3Bucket": "lambdafunctions", "S3Key": "javafunctions.zip"
    }, "Runtime": "java8", "Role": {
      "Fn::GetAtt": [
        "CartLambdaFunctionsRole", "Arn"
      ]
    }
  }, "DependsOn": [
    "Carts"
  ]
}, "ConfirmCartFunction": {
  "Type": "AWS::Lambda::Function", "Properties": {
    "Handler": "it.balsick.cloud.aws.thesis.cart.ConfirmCart::handler", "Code": {
      "S3Bucket": "lambdafunctions", "S3Key": "javafunctions.zip"
    }, "Runtime": "java8", "Role": {
      "Fn::GetAtt": [
        "CartLambdaFunctionsRole", "Arn"
      ]
    }
  }, "DependsOn": [
    "Carts"
  ]
}, "RemoveOfferFromCartsFunction": {
  "Type": "AWS::Lambda::Function", "Properties": {
    "Handler": "it.balsick.cloud.aws.thesis.offer.RemoveOfferFromCarts::handler", "Code": {
      "S3Bucket": "lambdafunctions", "S3Key": "javafunctions.zip"
    }, "Runtime": "java8", "Role": {
      "Fn::GetAtt": [
        "CartLambdaFunctionsRole", "Arn"
      ]
    }
  }, "DependsOn": [
    "Carts"
  ]
}, "RemoveFromCartFunction": {
  "Type": "AWS::Lambda::Function", "Properties": {
    "Handler": "it.balsick.cloud.aws.thesis.cart.RemoveFromCart::handler", "Code": {
      "S3Bucket": "lambdafunctions", "S3Key": "javafunctions.zip"
    }, "Runtime": "java8", "Role": {
      "Fn::GetAtt": [
        "CartLambdaFunctionsRole", "Arn"
      ]
    }
  }, "DependsOn": [
    "Carts"
  ]
}, "CheckTokenFunction": {
  "Type": "AWS::Lambda::Function", "Properties": {
    "Handler": "it.balsick.cloud.aws.thesis.user.registration.CheckToken::handler", "Code": {
```

```
      "S3Bucket": "lambdafunctions", "S3Key": "javafunctions.zip"
    }, "Runtime": "java8", "Role": {
      "Fn::GetAtt": [
       "AuthorizerLambdaFunctionRole", "Arn"
      ]
    }
   }, "DependsOn": [
    "Registrations"
   ]
  }, "LogInFunction": {
   "Type": "AWS::Lambda::Function", "Properties": {
    "Handler": "it.balsick.cloud.aws.thesis.user.registration.LogIn::handler", "Code": {
     "S3Bucket": "lambdafunctions", "S3Key": "javafunctions.zip"
    }, "Runtime": "java8", "Role": {
     "Fn::GetAtt": [
      "AuthorizerLambdaFunctionRole", "Arn"
     ]
    }
   }, "DependsOn": [
    "Registrations"
   ]
  }, "SignUpFunction": {
   "Type": "AWS::Lambda::Function", "Properties": {
    "Handler": "it.balsick.cloud.aws.thesis.user.registration.SignUp::handler", "Code": {
     "S3Bucket": "lambdafunctions", "S3Key": "javafunctions.zip"
    }, "Runtime": "java8", "Role": {
     "Fn::GetAtt": [
      "AuthorizerLambdaFunctionRole", "Arn"
     ]
    }
   }, "DependsOn": [
    "Registrations"
   ]
  }, "UpdateCartDomainSettingsFunction": {
   "Type": "AWS::Lambda::Function", "Properties": {
    "Handler": "it.balsick.cloud.aws.thesis.cart.UpdateCartDomainSettings::handler", "Code": {
     "S3Bucket": "lambdafunctions", "S3Key": "javafunctions.zip"
    }, "Runtime": "java8", "Role": {
     "Fn::GetAtt": [
      "PrivateCartLambdaFunctionRole", "Arn"
     ]
    }
   }, "DependsOn": [
    "DomainSettings"
   ]
  }, "UpdateCartDomainProductSettingsFunction": {
   "Type": "AWS::Lambda::Function", "Properties": {
    "Handler": "it.balsick.cloud.aws.thesis.cart.UpdateCartDomainProductSettings::handler",
"Code": {
     "S3Bucket": "lambdafunctions", "S3Key": "javafunctions.zip"
    }, "Runtime": "java8", "Role": {
     "Fn::GetAtt": [
      "PrivateCartLambdaFunctionRole", "Arn"
     ]
    }
   }, "DependsOn": [
    "ProductDomainSettings"
   ]
  }, "Carts": {
```

```
  "Type": "AWS::DynamoDB::Table", "Properties": {
   "AttributeDefinitions": [
    {
     "AttributeName": "user", "AttributeType": "S"
    }, {
     "AttributeName": "item", "AttributeType": "S"
    }
   ], "KeySchema": [
    {
     "AttributeName": "user", "KeyType": "HASH"
    }, {
     "AttributeName": "item", "KeyType": "RANGE"
    }
   ], "ProvisionedThroughput": {
    "ReadCapacityUnits": 1, "WriteCapacityUnits": 1
   }, "TableName": "Carts"
  }
 }, "DomainSettings": {
  "Type": "AWS::DynamoDB::Table", "Properties": {
   "AttributeDefinitions": [
    {
     "AttributeName": "settingKey", "AttributeType": "S"
    }
   ], "KeySchema": [
    {
     "AttributeName": "settingKey", "KeyType": "HASH"
    }
   ], "ProvisionedThroughput": {
    "ReadCapacityUnits": 1, "WriteCapacityUnits": 1
   }, "TableName": "DomainSettings"
  }
 }, "ProductDomainSettings": {
  "Type": "AWS::DynamoDB::Table", "Properties": {
   "AttributeDefinitions": [
    {
     "AttributeName": "product", "AttributeType": "S"
    }, {
     "AttributeName": "settingKey", "AttributeType": "S"
    }
   ], "KeySchema": [
    {
     "AttributeName": "product", "KeyType": "HASH"
    }, {
     "AttributeName": "settingKey", "KeyType": "RANGE"
    }
   ], "ProvisionedThroughput": {
    "ReadCapacityUnits": 1, "WriteCapacityUnits": 1
   }, "TableName": "ProductDomainSettings"
  }
 }, "Registrations": {
  "Type": "AWS::DynamoDB::Table", "Properties": {
   "AttributeDefinitions": [
    {
     "AttributeName": "username", "AttributeType": "S"
    }
   ], "KeySchema": [
    {
     "AttributeName": "username", "KeyType": "HASH"
    }
```

```
    ], "ProvisionedThroughput": {
      "ReadCapacityUnits": 1, "WriteCapacityUnits": 1
    }, "TableName": "Registrations"
  }
 }
}
}
```

7.1.5 AppSync Courier Authentication check-token

7.1.5.1 authenticate.js

```
const SECRET_JWT_KEY = "YwuF0wvvYwKJG4j3";
const jwt = require('jsonwebtoken');

const validate = token => {
  try {
    const decoded = jwt.verify(token, SECRET_JWT_KEY);
    console.log(decoded);
    return decoded.sub;
  } catch (e) {
    console.log(e);
    return null;
  }
};

module.exports = {validate: validate};
```

7.1.5.2 graphql-authenticate.js

```
'use strict'

const validate = require('./authenticate').validate;

exports.handler = async event => {
  console.log(event);
  const principal = validate(event.token);
  if (!principal)
    return {
      errorType: "AUTHORIZATION_ERROR",
      errorMessage: "Error with authorization token"
    };
  switch (event.field) {
    case "createBookReview":
    case "updateBookReview":
    case "deleteBookReview":
      let result = event.arguments;
      result.user = principal;
      return result;
    default:
      return "unable to resolve "+event.field;
  }
};
```

7.1.6 Java vs Node.js

Here they are presented the Java and the Node.js code used to test the cold start and the test, written in Java, that invokes the functions.

7.1.6.1 Java Lambda function

```
public class LogIn {

  public Result handler(LogInRequest request, Context context) {

    Registration registration;
    try {
      registration = DDBInterface.getItem(Registration.class, request.getUser());
    } catch (TooManyResultsException | ItemNotFoundException e1) {
      e1.printStackTrace();
      return new ErrorResult(ErrorType.BAD_REQUEST);
    }
    String salt = registration.getSalt();
    MessageDigest digest;
    try {
      digest = MessageDigest.getInstance(MessageDigestAlgorithms.SHA_256);
      digest.reset();
      digest.update(salt.getBytes());
      byte[] hashedBytes = digest.digest(Utils.stringToByte(request.getPassword()));
      String hashedPassword = new String(hashedBytes);
      if (!hashedPassword.equals(registration.getHashedPassword()))
        return new ErrorResult(ErrorType.UNAUTHORIZED);
    } catch (Exception ex) {
      return new ErrorResult(ErrorType.BAD_REQUEST);
    }

    Algorithm algorithmHS;
    try {
      algorithmHS = Algorithm.HMAC256(UserUtils.SECRET_JWT_KEY);
    } catch (IllegalArgumentException | UnsupportedEncodingException e) {
      e.printStackTrace();
      return new ErrorResult(ErrorType.UNSPECIFIED);
    }
    String token = JWT.create()
        .withIssuer("auth0")
        .withSubject(request.getUser())
        .sign(algorithmHS);
    return new GetTokenResult(token);
  }
}
```

7.1.6.2 Node.js function

```
const shipping_utils = require('../shipping-utils');
const utils = require('../utils');
const crypto = require('crypto');

exports.handler = async (event, context) => {
  let data = await utils.dynamodb.getItem({
    Key: {
      "courier": {
        S: event.courier
      }
    },
    TableName: "CourierRegistrations"
  }).promise();
  if (!data.Item)
    return {resultCode: 401};
  const courier = utils.ItemToValue(data.Item);
  const hmac = crypto.createHmac('sha256', courier.salt);
  hmac.update(event.password);
  if (hmac.digest('hex') !== courier.hashedPassword)
    return {resultCode: 401};
  const jwt = require('jsonwebtoken');
  const token = jwt.sign({ iss: 'auth0', courier: event.courier}, shipping_utils.SECRET_JWT_KEY);
  return {
    resultCode: 200,
    body: JSON.stringify(token)
  };
};
```

7.1.6.3 Java vs Node.js Test

```
import static java.util.concurrent.CompletableFuture.allOf;
import static java.util.concurrent.CompletableFuture.runAsync;

class LogInTests {

   private static String
         logInFunction =
         "aws lambda invoke --function-name UsedBookStoreThesis-LogInFunction --payload " +
               "\"{\\\"user\\\":\\\"girolamo\\\",\\\"password\\\":\\\"password\\\"}\" LogInFunction.txt";
   private static String
         courierLogInFunction =
         "aws lambda invoke --function-name UsedBookStoreThesis-CourierLogIn --payload " +
               "\"{\\\"courier\\\":\\\"ermenegildo\\\",\\\"password\\\":\\\"password\\\"}\" CourierLo-
gIn.txt";
   private static IntConsumer lifRunnable = (ignored) -> execute(logInFunction);
   private static IntConsumer clifRunnable = (ignored) -> execute(courierLogInFunction);

   public static void main(String[] args) throws ExecutionException, InterruptedException {
      for (int i = 0; i < 2; i++)
         sleepAndInvokeFunctions(15 * 60 * 1000, 300);
      for (int i = 0; i < 6; i++)
         sleepAndInvokeFunctions(2 * (6 - i) * 60 * 1000, 60);
   }

   private static void sleepAndInvokeFunctions(long millis, int totQty)
         throws ExecutionException, InterruptedException {
      for (int qty : new Integer[]{1, 5, 10, 15, 20}) {
         sleep(millis);
         allOf(getCompletableFutures(qty, totQty)).get();
      }
   }

   private static void sleep(long millis) {
      System.out.println("[" + new SimpleDateFormat("MM-dd HH:mm:ss").format(
            Calendar.getInstance().getTime()) + "] Started waiting for " + millis + "ms");
      try {
         Thread.sleep(millis);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }

   private static CompletableFuture[] getCompletableFutures(int parallelInstances, int totCount) {
      return IntStream.range(0, parallelInstances)
            .mapToObj(ignored -> new CompletableFuture[]{
                  runAsync(() -> IntStream.range(0, totCount / parallelInstances)
                        .forEach(lifRunnable)),
                  runAsync(() -> IntStream.range(0, totCount / parallelInstances)
                        .forEach(clifRunnable))})
            .flatMap(Arrays::stream)
            .toArray(CompletableFuture[]::new);
   }
   //[...]
}
```