Third Asia Pacific Conference on Advanced Research (APCAR, Melbourne, July, 2016)
ISBN:978 0 9943656 20
www.apiar.org.au

MICROSERVICES ARCHITECTURE FOR TOUR ACTIVITIES APPLICATION

Nuno Silva[a], Paulo Tomé[b]
[ab]Escola Superior de Tecnologia e Gestão de Viseu, Viseu, Portugal
Corresponding email: nuno_silva2010@live.com.pt

Abstract

Tour Activities Management Systems are currently an important issue in the Information Systems domain. In this paper, we show an architecture to manage tour activities in the tourism área. This architecture was developed based in the study of the most used software architectures. Based on this study of architectures, we decided to use the approach of independent services like microservices pattern. This pattern is used to architect large, complex and long-lived applications. This applications are composed with multiple microservices when each one of them is a lightweight and independent service that performs single functions and collaborates with other similar services. The term microservices strongly suggests that the services should be as small as possible.

Keywords: Architecture, Microservices, Tour Activities Management Systems.

1.Introduction

The Information Technology plays an important role for the tourists (Sigala, 2005; Huang & Zhu, 2011; Abdulhamid & Usman, 2014). Nowadays, there are a large set of Software Applications that can be used in the tourism domain. The concept of Tour Activity Management System was recently introduced in the Tourism domain.

Usually, an Tour Activity Management System comprises of several functions (Aloha Software, 2016), such as booking.

This paper proposes an architecture for Tour Activity Management Systems. Because microservices are new frameworks and consists of a flexible pattern, we consider that our proposal is straightforward.

The microservices pattern (Dmitry Namiot, 2014; Richardson, 2016) is the approach used for the application architecture. Microservices grew from Service Oriented Architecture (SOA), which gained popularity in the early 2000s and emerged as a way to combat large monolithic applications.

The powerful concept of microservices is gradually changing the industry of development webservices applications. This architecture concept is an approach to develop an application as a set of small independent and autonom services. Services that use some lightweight mechanisms to communicate and they are deployed absolutely independently.

This article describes how we use the increasingly popular microservices architecture pattern to develop a new application for the tourism area, specifically for tour activities companies. This application has similar goals to help companies to manage all related information like reservations, availability and payments.

Software Systems in Tour Activity domain

In the tour activity area, most of the systems that are currently available is based on monolithic architecture. This architecture is the common approach on the industry of software development. It is developed, tested, packaged, and deployed as a unique service. But, internally this architecture may have several services, components, etc.

With Monolithic application, it's not easy to understand and modify as the application is getting bigger. With the growing application, it is difficult to add new developers or to replace leaving team members.

Next, we enumerate several systems that was studied and analysed to come on a conclusion about the best architecture to choose for the application that will be created. The next comparison table is composed by the components of each referenced system.

Table 1: System Analysis

Component/System	Rezdy (2016)	Rezgo (2016)	TourCMS (2016)	Checkfront (2016)
Architecture	-	Monolithic	-	-
API	REST	REST	REST	REST
Response Formatting	JSON / XML	XML	XML	JSON
Authentication	API Key	API Key	-	OAuth2

The Application Architecture

Many organizations use the microservices and can better meet the needs of modern application development. A small change to a service can be committed, tested and deployed immediately since changes are isolated from the rest of the system. In the case of one service requiring extra performance, we just need to scale the individual service that needs additional capacity rather than scale all the system like in Monolithic systems. On microservices, it is possible to have small teams developing each service and have different programming languages on each service. Essentially, between services must be defined as a contract that will be shared by all development teams on what each service should provide to others.

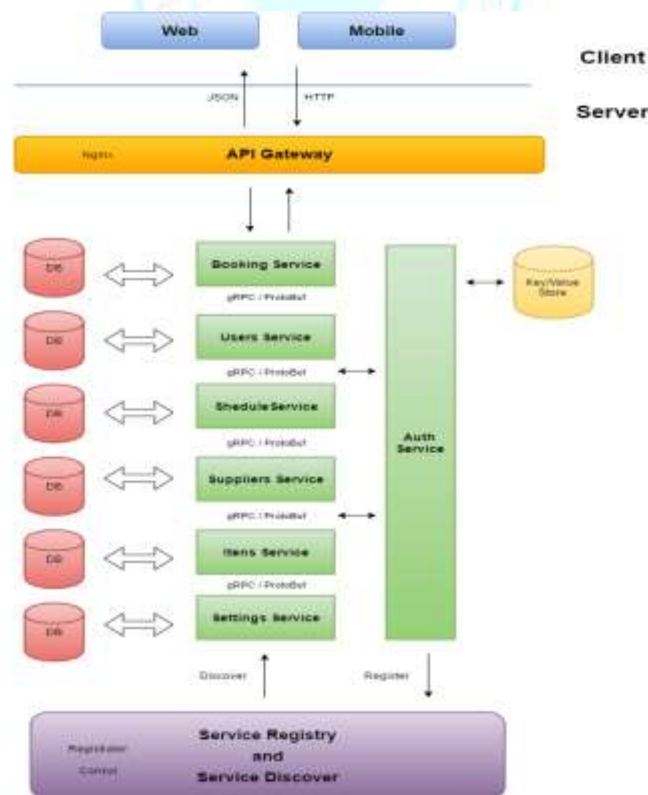


Figure 1: System Architecture

ISBN:978 0 9943656 20
www.apiar.org.au

In a general way, the application is divided into two layers, Client and Server. The Client layer is composed by all the platforms to perform the interaction with final user, in this case, the front end website and all mobile applications. On the Server side we have all the application logic based on the microservices pattern. This Layer is composed by multiple patterns like **Service Discovery and Register**, **API Gateway**, **Microservices Communication** and **Authentication**. This patterns help to understand the application infrastructure and distribution of each responsibility on the behaviour of all components and services.

All the infrastructure is supported by Docker (2016), one of the biggest companies in the technology industry today. In the official documentation, Docker was defined "an open platform for developing, shipping and running applications". In simple terms, this tool helps to deploy and run an application with all dependencies that it needs to run. That application is a single and isolated container that we can just throw at any server and it will simply work. A Docker container starts in the order of seconds. We can describe a Docker containers as lightweight VMs, but in reality they are much more than that. On VMs we can probably run just a few of them on a regular piece of hardware, with Docker it's easy to run dozens of Docker containers on the same piece of hardware.

Each small and autonomous service of the application is a Docker container and the set of all services is defined as a swarm. To deploy and build this swarm of independent services we use Docker Compose, defined by official documentation as "a tool for defining and running complex applications with Docker. With Compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running."

Service Discovery and Register

We are using Consul as a service discovery mechanism and Registrator to transparently register our containers there. Consul can be connected by multiple Clients that use Consul API to register new services, specifying a name and additional information in the form of tags. Then, by the same way, the client can query Consul for services that match their criteria. The main function of Registrator is ensuring that they are started on the same host where it is currently running, extracting information from them and then registering those containers on Consul server. It is designed to be run as an independent Docker container. With this approach, our containers are completely ignorant about how they will be discovered and about all infrastructure information.

API Gateway

The API Gateway is the entry point of all requests by clients that use the application, basically it works as a proxy. That proxy pass the request to the correct service and transmit the response back to client. To make this routing, we use Nginx with a configuration file generated by consul-template library. Consul template queries a Consul instance and updates a template which generate a Nginx configuration file that contains all the correspondent address and port for each service.

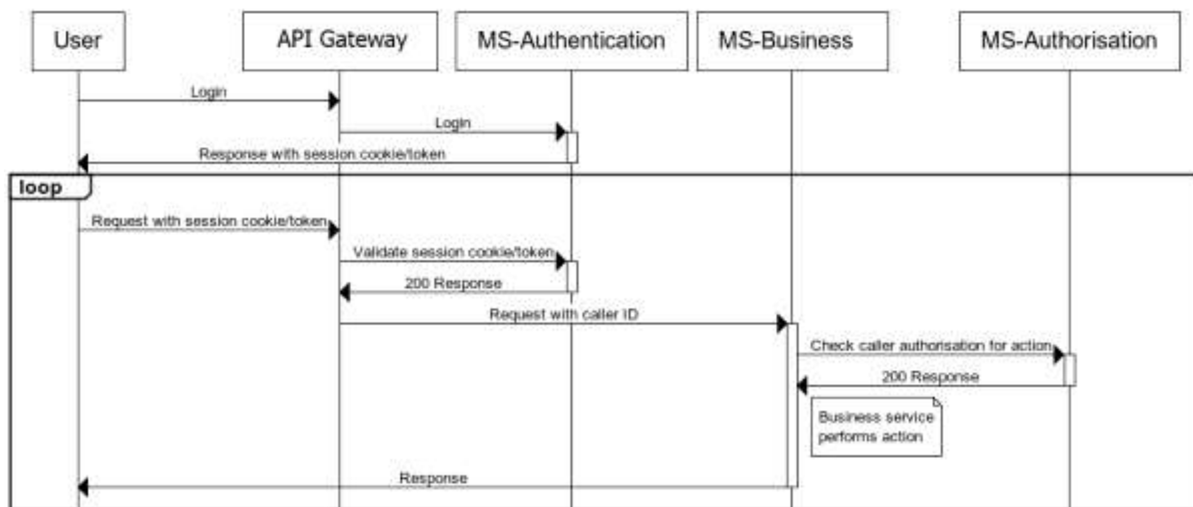
Microservices Communication

The communication between services is the biggest challenge in the development of application. One obvious approach is to partition services by use of case, and when each service communicate with others. Http protocol is the initial concept that we thought, but the biggest problem is the potential delays for remote calls. Then, we find gRPC, a HTTP/2 RPC Framework for Microservices. Google considers gRPC a "bandwidth and CPU efficient, low latency way to create massively distributed systems that span data centers, as well as power mobile apps, real-time communications, IoT devices and APIs.". Can be used in many scenarios, but gRPC is targeted primarily for microservices. A service interface and the data

types it handles is defined using the Protocol Buffers IDL, and with the help of compilers one can generate client and server stubs in 10 languages: C, C++, C#, Go, Java, Node.js, Objective-C, PHP, Python, Ruby.

Authentication

The authentication of the user on the system is made by a simple process. Basically, users request the authorization in the first instance by login and the authentication service returns a session token to the user. Then, on each request the user have to send the given token and then is validated by the authentication service. Next, the required service by the user is informed that user is authorized to perform the requested action and this service perform the action returning the response to the user. This process must be used for any action which need the authorization by the system.



Database

Each service has their own database with their own data and own structure. The relationships between databases is made by external referenced id's. With this approach, we have the flexibility to change the structure of one database without affecting the entire database system.
For some services with the probability of saving greatest volumes of data and make a biggest number of queries, we will create a Non Relational Database (NoSQL) like mongodb, for example. In the other side on database that save a small number of data, we plan use a relational database like Postgres SQL.

2. Conclusions and Future Work

In this article, we propose an architecture for Tour Activities Management System based on Microservices patterns. This architecture is extensible and can be adjustable for future developments.

The architecture is already implemented and is being tested in a real case situation. Regarding to the implemented system we can assure that the system is stable and can easily increase to other functions.

ISBN:978 0 9943656 20
www.apiar.org.au

References

i. Abdulhamid, S. M. & Usman, G., 2014. *Destination Information Management System for Tourist*.

ii. Aloha Software. 2016. *Activity Manager Software Screen Shot - Travel Industry Management Software for Tour Operators - Maui Hawaii Travel Industry Software*. Available at: http://www.alohasoftware.net/activitymanager.shtml [Accessed May 30, 2016]

iii. Checkfront HQ. 2016. *Checkfront Online Booking System & Reservation Software*. Available at: https://www.checkfront.com/ [Accessed May 30, 2016]

iv. Dmitry Namiot, M. S. S., 2014. On Micro-Services Architecture. *Int. J. Open Inf. Technol*, 2(9), pp. 24–27.

v. Docker, I., 2016. *Docker Compose*. Available at: https://docs.docker.com/compose/ [Accessed May 30, 2016]

vi. Huang, W. & Zhu, B., 2011. *Management of Tourism Group and Technology of The Personalized Tour Based on RFID*. In 2011 Chinese Control and Decision Conference (CCDC), pp. 975–978.

vii. Rezdy. 2016. *Online Booking Software for Tour Operators*. Available at: https://www.rezdy.com/ [Accessed May 30, 2016]

viii. Rezgo. 2016. *Rezgo | Powering Tour & Activity Businesses Worldwide*. Available at: https://www.rezgo.com/ [Accessed May 30, 2016]

ix. Richardson, C., 2016. *Microservices Architecture Pattern*. Available at: http://microservices.io/patterns/microservices.html [Accessed May 30, 2016]

x. Sigala, M., 2005. New Media and Technologies: Trends and Management Issues for Cultural Tourism. In *International Cultural Tourism*. Elsevier, pp. 167–180.

xi. Travel UCD Limited. 2016. *TourCMS - Tour Operator Reservation System and Website CMS*. Available at: http://www.tourcms.com/ [Accessed May 30, 2016]