Online Ticketing

Beau De Clercq

January 10, 2020

Abstract

Requirements

- A user will need to register at least 2 weeks before the tickets go on sale.
- In the event that there are no available tickets left, the user should immediately get a message stating that the event is sold out.
- The whole action of buying tickets should take no more than ten minutes.
- In the final product some form of load balancing should be present to ensure the system can handle peak loads.
- The banking service should verify whether the user has sufficient funds left on their account.
- All communication should happen in a secure way by means of messages encrypted by a user-specific key which is stored in a secure keyvault.

1 Assumptions

For this project a few assumptions were made:

- During registration, the encryption key for each user was immediately stored in the keyvault such that when the tickets will go on sale this keyvault can be distributed to the other services and thereby reducing the potential bottleneck.
  We believe that this will not compromise the security requirements due to the fact that all services should be equally protected against potential attacks.

2 Simulations

2.1 Design

In order to make a simulation of the eventual system, we used the ABS modeling language to verify whether the system would be able to satisfy the different requirements.

In a first step the necessary services needed to be composed. After some deliberations we settled on following list of core services:

- a Users service that will be responsible for the registration and eventual authentication of users,
- an Interface service which will act as the interface between the user and the ticketing service,
- a Banking service to simulate the times a user would have to wait while his or her payment is being executed,
- a Tickets service which acts as database to keep track of the amount of tickets that are still available.

Additionally we added a load balancer based on the Round Robin principle in order to verify whether our model would be able to handle peak loads.
In this implementation we made the assumption that each of the individual services has its own copy of the keyvault.

2.2 Calibration

After we composed all of our services we decided to add some calibrations concerning timings.
For instance, not all banks will have the same response time. In order to try and simulate this fact we added three different time values in which the banking service will respond.
When running the system with different configurations we the results shown in Table 1 and Table 2.
From these tables it is clear that, in order to let this project succeed, we need to let go of the synchronous approach and focus on the asynchronous one. Another conclusion is that when the bank transaction times are relatively close to each other, the impact they have on the overall run-time is rather limited.
A small remark to the results is that only requests from the "tickets available" scenario are shown, the ones for the "no tickets available" scenario are handled instantly.
Table 2 clearly show that we can keep the total

Bank	Transaction time in seconds	Total request time in seconds
X	1	6.100
	0.1	0.700
	0.01	0.160
Y	2	8.100
	0.2	0.900
	0.02	0.180
Z	5	15.100
	0.5	1.600
	0.05	0.250

Table 1: Influence of transaction times in a synchronous system (maximum 200 requests).

handling time for all request reasonably low by going for a combination of low bank transaction times (which need to be negotiated) and a high number of servers.

2.3 Meeting the requirements

The first requirement of the system is that a user should never have to wait for more than 10 minutes. To this end there should be some sort of load balancing in place for the system to handle peak-loads.
As shown in Table 2, the number of servers needed depends for the largest part on the number of available tickets and the amount of expected requests to occur at the same time. From the same data a graph can be plotted which will show that the amount of time needed to handle all incoming requests lowers exponentially with the number of servers.
Since the amount of memory in the machine used to run the tests is limited, we leave the calculations as for the exact amount of servers that would be required to the reader as we provide the code in Appendix D. While trying to find the answer ourselves we reached our limit simulating 10,000 requests with 100 available servers, resulting in a total request time of 6.2 seconds assuming bank transaction time of 0.5 seconds.

The second requirement of the system is that all communication should be encrypted. To this end we propose to put a copy of the central keyvault in each of the services. This is possible because of the fact that all users should have registered a couple of weeks in advance. By doing so we can ensure that encryption and decryption both have an insignificant impact on the total request time as this would eliminate sending various requests to the central keyvault service.

The final requirement is that a user's balance should be checked to verify that he/she has sufficient funds. In case the user pays using a credit card, this requirement is useless. People often decide to pay using a credit card when the balance of their account shouldn't be considered, it may be the case that they decide to buy tickets at a time when they don't have sufficient funds yet. The credit card then, as the name suggest, provides the necessary credit. How and when this debt between the user and credit card company is settled doesn't belong to the scope of this project.
In the case a user decides to pay using a debit card, this credit check is implicitly executed and therefore already accounted for in the simulation.

3 Proof of concept

3.1 Design

To build a proof of concept for this project we decided to use the Docker framework in order to containerize our final solution. After reviewing our initial design we decided to keep all of the elements:

- a Users service that will be responsible for the registration and eventual authentication of users,
- an Interface service which will act as the interface between the user and the ticketing service,
- a Banking service to simulate the times a user would have to wait while his or her payment is being executed,
- a Tickets service which acts as database to keep track of the amount of tickets that are still available.

Again we assume that each of the individual services has their own copy of a central keyvault that has been filled during the registration period. In order to achieve the required load balancing the application can be launched using Kubernetes.

Amount of servers	Amount of tickets	Number of requests	Bank	Transaction time in seconds	Total request time in seconds
1	10000	1000	X	1	55.350
				0.1	50.600
				0.01	50.110
			Y	2	56.100
				0.2	50.050
				0.02	47.500
			Z	5	60.100
				0.5	51.100
				0.05	50.200
2	10000	1000	X	1	30.100
				0.1	25.600
				0.01	25.110
			Y	2	31.150
				0.2	25.700
				0.02	25.130
			Z	5	35.100
				0.5	26.100
				0.05	25.200
3	10000	1000	X	1	21.850
				0.1	17.300
				0.01	16.810
			Y	2	22.900
				0.2	17.500
				0.02	16.880
			Z	5	26.800
				0.5	17.800
				0.05	16.900
4	10000	1000	X	1	17.650
				0.1	13.100
				0.01	12.610
			Y	2	18.600
				0.2	13.200
				0.02	12.630
			Z	5	22.600
				0.5	13.600
				0.05	12.700

Table 2: Influence of transaction times in an asynchronous system.

3.2 Calibration

We decided to calibrate our system using a random bank transaction time of maximum 5 seconds, all other timing values are now implicit when using the actual system.

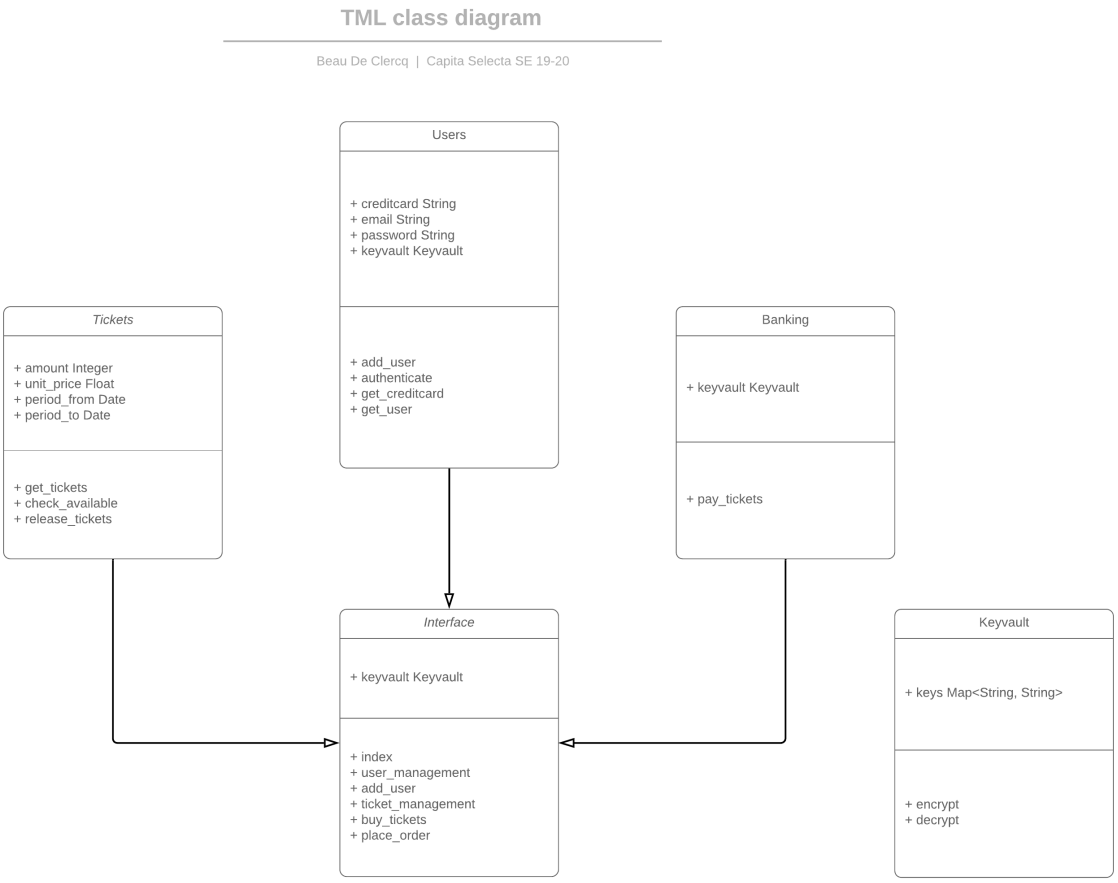
3.3 Meeting the requirements

4 Can it be build?

5 Next steps

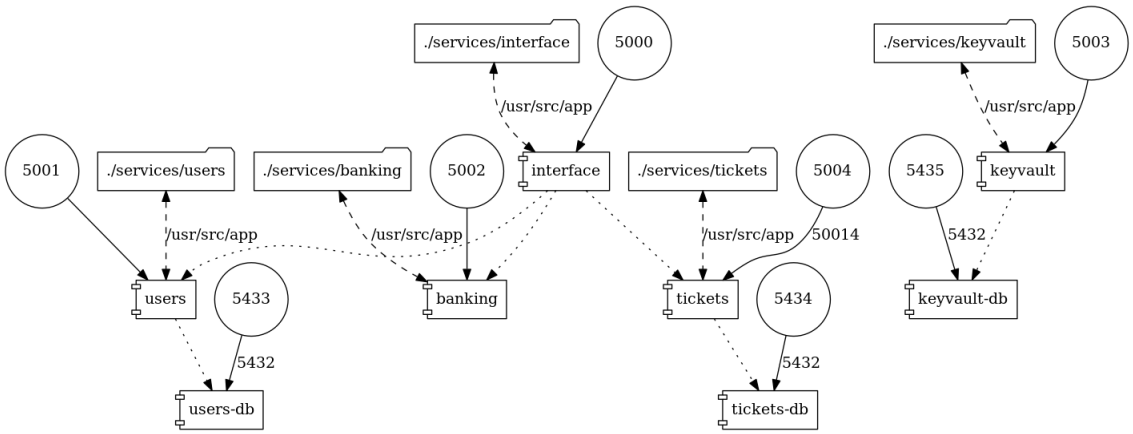
Appendices

A Class diagram



B Deployment diagram

Generated by running following command in the folder containing the docker yml file:
`docker run --rm -it --name dcv -v “(pwd):/input” pmsipilot/docker-compose-viz`



C Sequence diagrams

D ABS code

```
// duration(x, y) means waiting at least x miliseconds and at
      most y miliseconds

module TML;
import * from ABS.DC;

interface LoadBalancer {
  Unit addWorker(Shop s);
  Shop getWorker();
  Unit releaseWorker(Shop s);
}

class RoundRobinLoadBalancer() implements LoadBalancer {
  List<Shop> available = Nil;
  List<Shop> inuse = Nil;

  Unit addWorker(Shop s){
    available = appendright(available,s);
  }

  Shop getWorker(){
    await (available != Nil);
    Shop s = head(available);
    available = tail(available);
    inuse = appendright(inuse,s);
    return s;
  }

  Unit releaseWorker(Shop s){
    available = appendright(available,s);
    inuse = without(inuse,s);
  }

}

interface User {
  Bool register();
  Bool authenticate();
  Bool tickets();
  String getName();
  String getEmail();
  String getCCtype();
}

class User(String name, Int age, String cctype, String email,
      Keyvault k) implements User {
  Bool register() {
    await duration(10, 99/2);
    println("Registered " + name);
    return True;
  }

  Bool authenticate(){
    await duration(10, 59/2);
    return True;
  }

  Bool tickets(){
    await duration(30, 99/2);
    println("Printing tickets");
    return True;
  }

  String getName() {
    k.encrypt();
    return name;
  }

  String getEmail() {
    k.encrypt();
    return email;
  }

  String getCCtype() {
    k.encrypt();
    return cctype;
  }

}

interface Shop {
  Bool buyTickets(Int amount, User u, String date1, String date2)
       ;
}

class Shop(Bank b, Keyvault k, Tickets t, Duration responseTime)
      implements Shop {
  Bool buyTickets(Int amount, User u, String date1, String date2)
       {
    Bool status = True;
    if (amount>4){
    println("Too many tickets ordered, abort");
    status = False;
  }

  Bool ticket_status = t.checkAvailable(date1, date2, amount);
    //println(toString(ticket_status));
    if (ticket_status == True){
      String cctype = await u!getCCtype();
      Bool temp = k.decrypt();
      Bool payment = await b!process(42, cctype);
      if (!payment){
        status = False;
        t!releaseTickets(date1, date2, amount);
      }
      else{
        println("Tickets have been successfully booked");
      }
    }
    else{
      println("No ticket available");
      status = False;
    }
    println(toString(now()));
    return status;
  }
}

interface Bank {
  Bool process(Int price, String cctype);
}

class Bank(Keyvault k) implements Bank {
  Bool process(Int price, String cctype){
    Bool status = True;
    if (cctype == "visa"){
      await duration(200, 800);
    }
    else if (cctype == "maestro"){
      await duration(500, 1500);
    }
    else if (cctype == "mastercard"){
      await duration(100, 600);
    }
    return status;
  }
}

interface Keyvault {
  Bool encrypt();
  Bool decrypt();
}

class Keyvault() implements Keyvault {
  Bool encrypt(){
    await duration(40, 50);
    return True;
  }

  Bool decrypt(){
    await duration(40, 50);
    return True;
  }
}

interface Tickets {
  Bool checkAvailable(String date1, String date2, Int amount);
  Bool releaseTickets(String date1, String date2, Int amount);
  Bool addTickets(String date1, String date2, Int amount);
}

class Tickets() implements Tickets{
  Map<Pair<String, String>, Int> tickets = map[];

  Bool checkAvailable(String date1, String date2, Int amount){
    Bool status = True;
    Pair<String, String> period = Pair(date1, date2);
    Int available = lookupDefault(tickets, period, 0);
    if (amount >= available) {
      status = False;
    }
    else{
      Int temp = available;
      available = temp-amount;
      tickets = put(tickets, period, available);
      status = True;
    }
    return status;
  }

  Bool releaseTickets(String date1, String date2, Int amount){
    Pair<String, String> period = Pair(date1, date2);
    Int available = lookupDefault(tickets, period, 0);
    Int temp = available;
    available = temp+amount;
    tickets = put(tickets, period, available);
    return True;
  }

  Bool addTickets(String date1, String date2, Int amount){
    Pair<String, String> period = Pair(date1, date2);
    Pair<Pair<String, String>, Int> p = Pair(period, amount);
    tickets = insert(tickets, p);
    return True;
  }
}

{
  CloudProvider p = new CloudProvider("TML");
  await p!setInstanceDescriptions(
  map[Pair("T2_MICRO", map[Pair(Memory,1), Pair(Speed,1)]),
  Pair("T2_SMALL", map[Pair(Memory,2), Pair(Speed,1)]),
  Pair("T2_MEDIUM", map[Pair(Memory,4), Pair(Speed,2)]),
  Pair("M4_LARGE", map[Pair(Memory,8), Pair(Speed,2)])]);

  DC  server1 = await p!launchInstanceNamed("T2_SMALL");
  DC  server2 = await p!launchInstanceNamed("T2_SMALL");
  DC  server3 = await p!launchInstanceNamed("T2_SMALL");

  DC  server4 = await p!launchInstanceNamed("M4_LARGE");
  DC  server5 = await p!launchInstanceNamed("M4_LARGE");
  DC  server6 = await p!launchInstanceNamed("M4_LARGE");
  DC  server7 = await p!launchInstanceNamed("M4_LARGE");

  [DC: server4] Keyvault kv = new Keyvault();
  [DC: server5] Bank bank = new Bank(kv);
  [DC: server6] LoadBalancer lb = new RoundRobinLoadBalancer();
  [DC: server7] Tickets t = new Tickets();

  Int nmbrTickets = 10000000; // This values can be changed for
        testing purposes
  t!addTickets("01-07-2020", "03-07-2020", nmbrTickets);

  Int nrWorkers = 100;
  while(nrWorkers > 0){
    Fut<DC> fs = p!launchInstanceNamed("M4_LARGE");
    DC vm = fs.get;
    [DC: vm] Shop shop = new Shop(bank, kv, t, Duration(1000));
```

```
      lb.addWorker(shop);
      nrWorkers = nrWorkers-1;
    }

    Int nrJobs = 10000; // Number can be changed for testing
          purposes
    println(toString(now()));
    while(nrJobs>0){
      Fut<Shop> s = lb!getWorker();
      User u = new User(toString(nrJobs), 22, "mastercard", "email"
            , kv);
      Shop shopworker = s.get;
      shopworker!buyTickets(4, u, "01-07-2020", "03-07-2020");
      lb.releaseWorker(shopworker);
      nrJobs = nrJobs - 1;
    }

    println(toString(now()));
    println("DONE");
  }
```