Electronic Shopping (E-Shopping)

Cloud Computing Course Project, *E-Shop Phase 1*
Submission Deadline: *Dey 21st, 1398*

Introduction

It has been a while since we were first introduced to online shopping. It has currently fit into our lives in an inseparable fashion. There used to be a time that the issue of trust would stop you from buying things online, but fortunately those problems have faded away. Nowadays, there are many online shops, and companies compete over the clients. Clients' needs have evolved through the past decade and now they consider many things before they make a purchase. Companies need to take actions to win clients over. For instance, they'd throw a sale where people could buy what they need with a discount at specific times of the year like Christmas, Black Friday, Yalda, Norouz and so on. At these times, people would feel that there is a competition and they need to move faster and make their purchase before the store runs out of them. This would introduce new challenges to the software development part of the process. At this point, web and application developers needed to make note of the situation and emphasize on the availability and scalability of their product; since any mistake in this regard could lead to losses that employers cannot take. Let us all do what we can to have a scalable online shop and hope for an uneventful Sale next time.

Outline

The MSA-Shop consists of four modules; *User Interface, Authentication*, *Account Management*, *Trade Management*. Here's how these modules interact with each other.

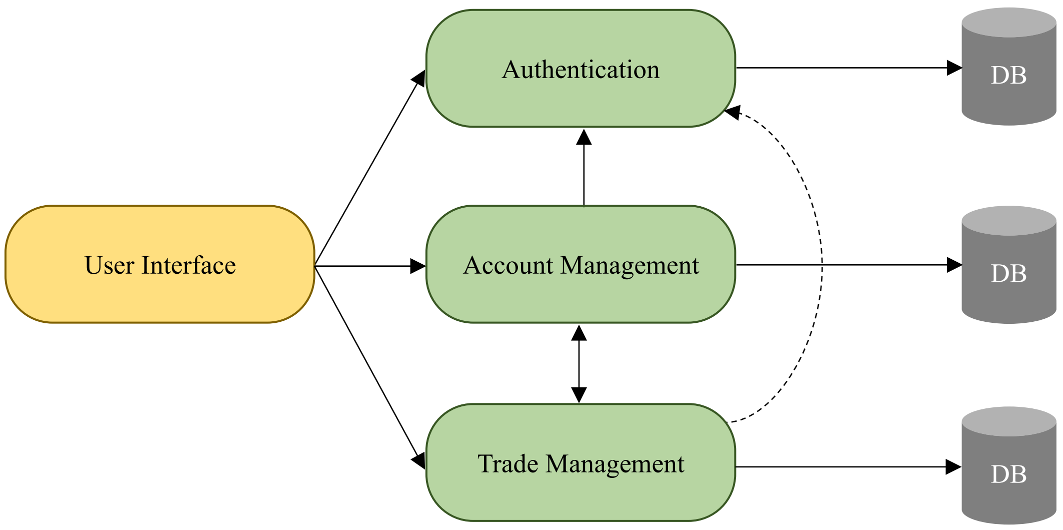

Figure 1 – **MSA-Shop Microservices Architecture**

The MSA-Shop is architected in a way similar to *eBay*. In this simple e-shop, each user can be a seller or buyer *at the same time*. Users can publish items to be sold or add items to their ***cart*** and finalize a purchase. Each user has a ***wallet*** which is charged in 2 ways. The user either sells an item and receive its share from the deal, or pays up through *ZarinPal*. In order for a user to make a purchase, he/she *has to have the purchase amount* in the wallet.

Each module is accountable for an *independent* category of functionalities and their communication will be made only through ***RESTful HTTP APIs***. Let's have an overview of what each module is responsible for:

- **User Interface:** All *user interaction* with the system will be only through the *User Interface* module. This module communicates with other modules to make what the user wishes come true (of course through API calls).
- **Authentication:** This module is responsible for *identity management*. User *signs up*, *signs in* and has his/her *session* managed through this module. When other modules need to check the *authority* of a command, they come to the *Authentication* module.
- **Account Management:** This module creates a *profile* and a *wallet* for each user and *manages* all interactions with these two entities.
- **Trade Management:** Everything related to what a user *buys* and *sells* are placed here.

How to Go About It

You need to write the code for the ***"Account Management"*** module of the E-Shopping system; dockerize it and push the image to the *Docker Hub* (automatically). Code language, framework and database will be your choice and do not matter, as long as they provide the functionalities described in this document. It is necessary to automate the process of building image from source code (take a look at this link). You also need to provide a ***docker-compose*** file (pushed alongside your code and *Dockerfile* to your git repository) that works on a *single* machine and integrates your module and its database to the Authentication Service. You do not need to dockerize this module. It has already been done and you can access it via this link. This module requires its own MongoDB (for further info, refer to the repository ReadMe file). Please note that the rest of the modules that make the system will be given to you for the next phase of the project and you will deploy the whole E-Shopping system as a cluster using Docker Swarm (you may be required to modify your microservice to match the whole system for the next part).

Description

Since you are supposed to implement the *Account Management* microservice, let's dig a little deeper and sort out some details.

Here are the *suggested* models to implement this microservice:

- Profile: *id*, *email*, *name, phoneNo, nationalCode, address, postalCode*
- Wallet: *id*, *profileID*, *value*
- Transaction: *id*, *profileID*, *createdAt*, *modifiedAt*, *amount*, *orderID*, *statusCode, refID*

Here are the APIs that you are ***required*** to implement:

- Get Heartbeat
  GET /account/heartbeat ()
  RETURNS → message: "Account Management is up and running"
- Create a Profile
  POST /account/profile (*email*, *password*, *name, phoneNo, nationalCode, address, postalCode*)
  RETURNS → token
- Update a Profile
  PUT /account/profile (*name*, *phoneNo*, *nationalCode*, *address*, *postalCode*)
  RETURNS → Profile
- Get a Profile
  GET /account/profile ()
  RETURNS → Profile
- Get a Wallet Value
  GET /account/wallet ()
  RETURNS → value
- Make a Payment
  POST /account/pay (*orderID*)
  RETURNS → *redirection to ZarinPal*
- Payment Callback
  GET /account/pay/callback () Query Parameters (*authority*, *transactionID)*
  RETURNS → *nothing*
- Get List of Transactions
  GET /account/transaction ()
  RETURNS → Transaction []

Notes:

- Underlined parameters are required.
- Modify the models in any way you'd prefer. ***You are obliged to provide the requested functionalities (APIs), not the data structure.***

- All of the requests except "Get Heartbeat", "*Create a Profile*" and "*Payment Callback*" require authorization.
- Every request that needs to be authorized should contain a bearer Token in its header.
- Since you do not have the Trade Management module yet, use mock data when necessary.
- In order to implement payment process and integrate with ZarinPal, check out their documentation and lab (to get the library).

Extra Credit

You may perform the following task to earn extra credit:

- ✓ You are already aware of how much *testing* is important in software development.
  Write tests for the ***"Account Management"*** module that you developed. Make sure that your test works fine and use docker to check if each commit does not break the code after building; **in that case only**, push the image to *Docker Hub*. This process is called ***automated testing*** (Hint).

Submission

Students need to pair for this project. Groups of only ***two*** people are allowed; you may try to do this without a teammate but I certainly advise you ***against*** it. Each group is required to create a repository on GitHub and invite my account *"SayidHosseini"* as a *Collaborator*. Your repository should contain a *ReadMe* file, explaining everything about your service (what it does, what are the dependencies, how to run the project, API doc and necessary references). Your project needs to be ready on Dey 21st, 1398 at 12 P.M. It is necessary for *each* student to commit changes via his/her own GitHub account while developing this module. *Commits will be reviewed!*

Please note that the project will not be accepted after the deadline and that the deadline will not be extended!

Good Luck!

SayidHosseini
SayidHosseini

References

- https://docs.docker.com/docker-for-windows/
- https://docs.docker.com/install/linux/docker-ce/ubuntu/
- https://docs.docker.com/get-started/
- https://docs.docker.com/compose/
- https://github.com/makbn/docker_basics_tutorial