Micro Service Me Up – Cloud Services Development with Microservices Architecture

https://github.com/TanqEdPublic/Micro-Service-Me-Up
Demo: https://youtu.be/8Ow0CRWHyyU

Tangqi Feng
Eduards Vagals

B. Sc.(Hons) in Software Development

Submitted on April 17, 2018

Final Year Project

Advised by: Dr. Brian McGinley

Department of Computer Science and Applied Physics
Galway-Mayo Institute of Technology (GMIT)

Table of Contents

Table of Illistrations

Chapter 1: Introduction

This dissertation has been prepared as a part of a learning process in Applied Project and Minor Dissertation module in B. Sc. Honours Degree in Computing & Software Development, GMIT to give an insight of developing Cloud-based web services using Microservices architecture style for individual service development, deployment and maintenance.

1.1 Aims and objectives

The aims of this dissertation are:

- Research in the area of Cloud web development and Microservices
- Gain practical experience in using trending software development methodologies that are suitable for Microservices development
- Gain an insight and a better understanding the various technologies that are associated with the development of Cloud-based web services
- Designing an appropriate architecture for Cloud service
- Undertake comprehensive research in full stack development, including database management, back-end and front-end development
- Finding solutions for problems encountered throughout a development cycle
- Research in automation technologies to speed up development and production of services
- Evaluate design and application throughout development iterations
- Draw a conclusion on developing Microservices, critically evaluating benefits and drawbacks of using architecture, and set recommendations for the new or further development

1.2 Why it is relevant

Over the past decade, the internet became available almost in all parts of the globe, used by many different devices from extra small embedded devices to large supercomputers, connecting billions of people. Production and usage of data over the internet showed significant growth rate comparing to 5 years ago, transforming it into a Big Data.
As software developers, we have to learn how to use and develop new techniques to work with Big Data, providing continues, reliable and fast services across different hardware, software environments and geographical locations. Thanks to a technological revolution, many ideas became possible to realise for practical use, and this flow of ideas is getting larger. These ideas bring big changes to development processes used in the past. This so-called “legacy systems” are harder to adapt to a rapid pace of innovation in Technological Industry. Software developers require learning how to face such changes to provide continues delivery of services, extend the system with new features and products rapidly, without major or critical side effects to the overall system.

How can Cloud services and Microservices architecture can help to address this issues?
Cloud services ensure that services are available to a user anywhere with basic internet connection. It can be a static web files server to host a web browser client, a full application that had to be installed to your desktop before, or a small neural network that can be fed with data to produce analytical reports. Consider that eventually everything will be moved to cloud to keep client devices clean from installing software that takes space and time.

> Microservices architecture is defined as a system composed of small autonomous services that worked together, modelled around business requirements. Each service is a single logical component that runs independently in it’s own process. Large service that covers big spec of requirements can be also composed of smaller components for ease of development and testing, but practicaly services are kept small, responsible for the execution of small logic of a system.

What are the benefits of developing Cloud services using Microservices architectural approach?
There are numerous benefits for picking Microservices approach, as well as, pitfalls you can get stuck in. We will discuss it more details in the following sections, but main benefits would be:

- Ability to make different choices in developments approach and technological stack
- Limitless possibilities in extending, adopting and improving system
- Complements Agile Methodology development

Although businessmen are looking for cons against prons to analyse risks involved in committing to something new and unexplored, it is better to discuss upfront if microservices architecture is suitable for the new project or is adaptable to legacy systems. It is often enough appears to be too steep of a learning curve for small projects with little business requirements, which may be too expensive and time-consuming.

Architecture is Agile friendly, especially fits well with Scrum processes. DevOps culture is playing an important part of developing services as each service would require a full stack development team assigned to it that takes ownership of a service from development to production and maintenance.

1.3 Dissertation brief

In the following dissertation paper we are going to discuss:

- ***Microservices Architecture Principals*** – the review of eight common principals of developing cloud services using the microservices architecture based on a talk given by *Sam Newman* on Development of Microservices.

    - Modelling around business domain
    - Culture of Automation
    - Hiding implementation details
    - Decentralised development
    - Independent deployment
    - Consumer first
    - Isolation of failure
    - High observability

- ***Development Methodology*** – description of methodology and project management processes used throughout the development of *Micro Service Me Up* project.

- ***Technology review*** – insight of technological stack used is development of *Micro Service Me Up* project, what it can do and why it was picked for the current project. We are also comparing and contrasting our choices with alternative available technologies.

- ***System Design*** – in this chapter we are introducing *Micro Service Me Up* architecture design we put together with most modern technological stack currently used in IT and software industry. It also covers the individual description of each service developed.

- ***System Evaluation*** – appraisal of the system developed up to date of the submission, description of an encountered problems and it's solutions, list of unsolved problems and what measurements to take to continue developing the project.

- ***Conclusion and Recommendations*** – the chapter about our impressions and understanding of the subject of research. List of recommendations for developing a new project using a similar approach and what should be done differently.

Here is a recap of how microservices differ from monolithic applications (all-in-one).

1.4 Monolithic Design

- **All business requirements in a single process**
- **Enterprise monoliths consist of 3 parts:**
  - **User UI**
  - **Business logic**
  - **Database layer**
- **Horizontal Slicing approach**
  - **Development teams split into speciality groups**
- **Horizontal Scaling**
  - **Monolith scales by replicating full application and deploying it into the independent process**
  - **Requires load balancing**

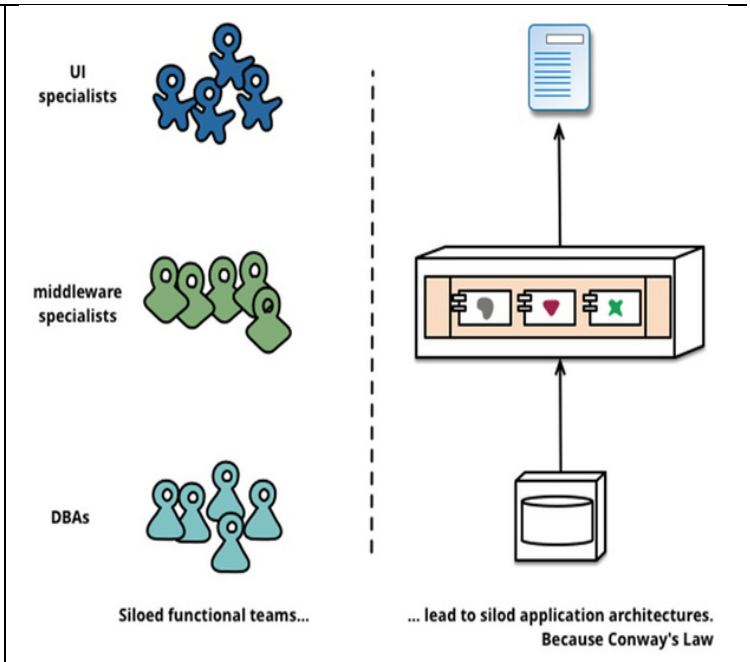


Figure 1 monolithic design

1.5 Microservices Design

- **Application functionalities built and deployed as a separate process.**
- **Each service is developed and managed by a full stack team. Introducing DevOps culture.**
- **Lightweight communication between services via HTTP protocol. RESTful Services.**
- **Vertical Slicing approach**
  - **Splitting of application logic into independent components.**
  - **Assigning full stack team for each component**

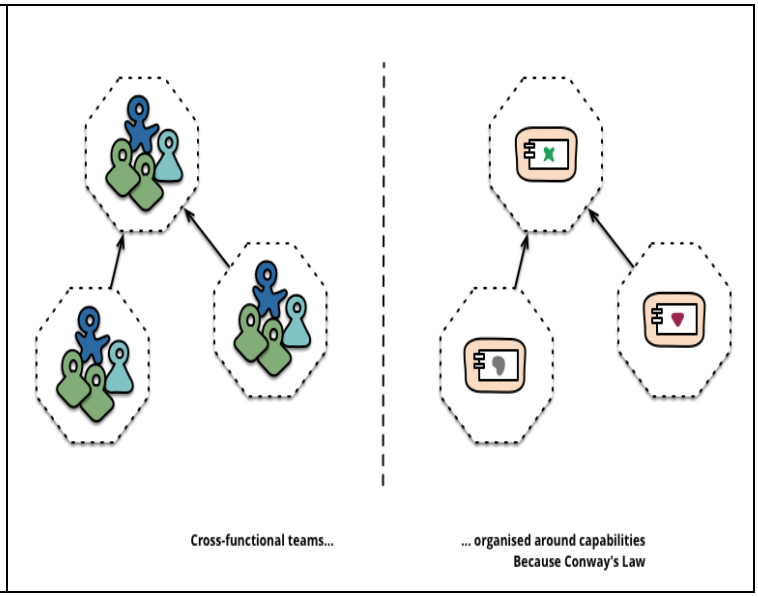


Figure 2 microservices design

Chapter 2: Microservices Architecture Principals

To build robust and flexible Microservices Ecosystem, it is sensible to follow certain patterns and principles. If taken into consideration, it will be easier to develop, extend and maintain software. In this chapter, we are recapping eight principles introduced and described by *Sam Newman* in his book *"Building Microservices"* [1] to familiarize the reader with concepts we used to develop out project applications. Techniques Sam describing in his book are used day to day in modern software industries where large-scale applications are common to develop.

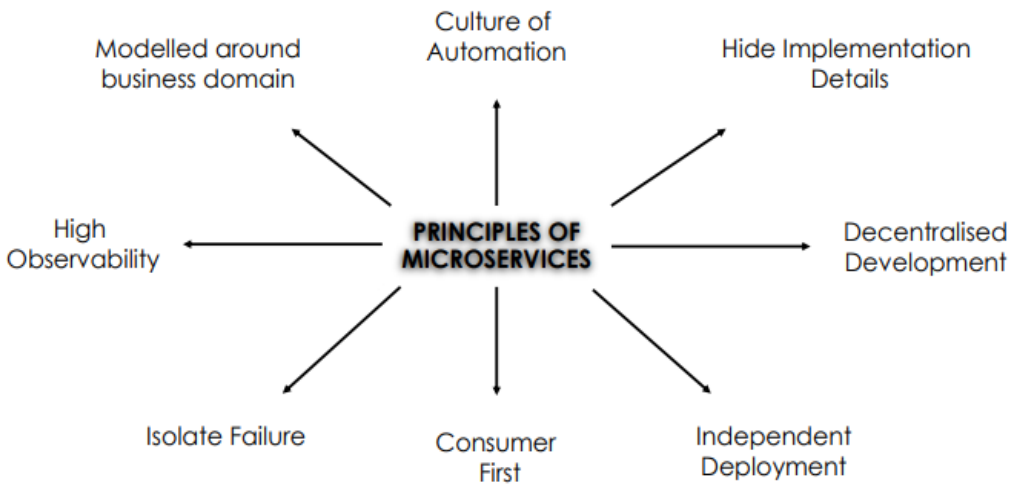


Figure 3 microservices architecture principals

2.1 Modelled On Business Domain

In software engineering, a domain model is a formal representation of the domain knowledge available to request by other peace of software. The business domain describes the requirements for which domain model is designed. Each microservice represents individual business domain to form an API that defines the contract between service and consumer. We can look at it as we go through the market, one tent sells only apples, pears and oranges, this is the contract this tent follows with respect to the market and customers. If we want to buy oranges we go there, if we want to buy potato or carrot we go to another tent that by definition of its contract sells vegetables.

It is the responsibility of the microsevice to negotiate a transaction with the customer and it is free to reject the transaction if a customer doesn't follow it's part of the contract. When going to an Irish market, make sure you bring euros and not British pounds, or you won't get served. In web development, the customer would send HTTP request to a service URL that in order to be processed should follow specific standard this service accepts, i.e., correct HTTP request method, required headers and appropriate body format. In response, service should follow it's part of the contract and return the correct product as described by the business model otherwise, customer also won't be happy and will complain!

The diagram below illustrates how the market can incorporate different services together to please its customers.

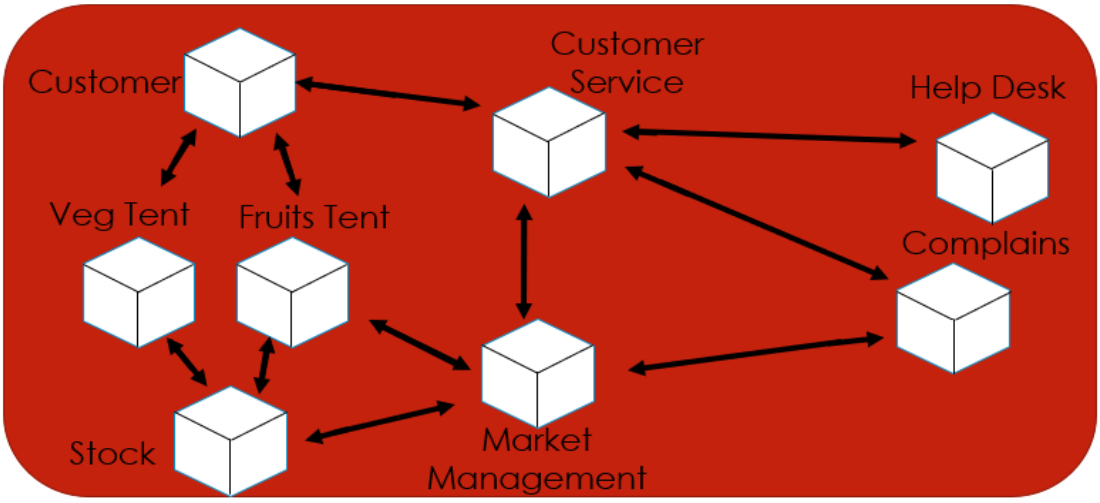


Figure 4 how market can incorporate different services together

Prior initiating development of services, teams should put well though out business models and have an agreement to follow these models. With Agile iterative processes, it is possible to change the description of business models, add new or remove redundant models as long as communication between development teams is well organized. An incorrect approach in defining business models directly reflects the violation of customer requirements for the given project. If discovered late in time may cost dearly in respect to time and effort of redesigning services, consequently doing financial loss to the company developing the product.

2.2 Culture of Automation

Continuous product delivery can be maintained if the development team is highly capable of automating monotonous day-to-day work. In small projects where only three to five services are involved, automation is not that viable. Developer can deploy and maintain that amount with ease. Automation becomes needed like an air on large scale projects, to prevent developer choking from the number of services to deploy or restart. Developers should be able to deploy components to dedicated virtual environments within few command lines. If it takes a lot of effort to do that to one service, imagine microservices systems with 60 or so services. Testers should also consider automation for testing basic service requirements that follows other services, as well as a test suit of its own capabilities. We always have to take into account that service internal structure can change, but should still pass test suit for external calls to this service. This would also note a first principle, *Model on the business domain*.

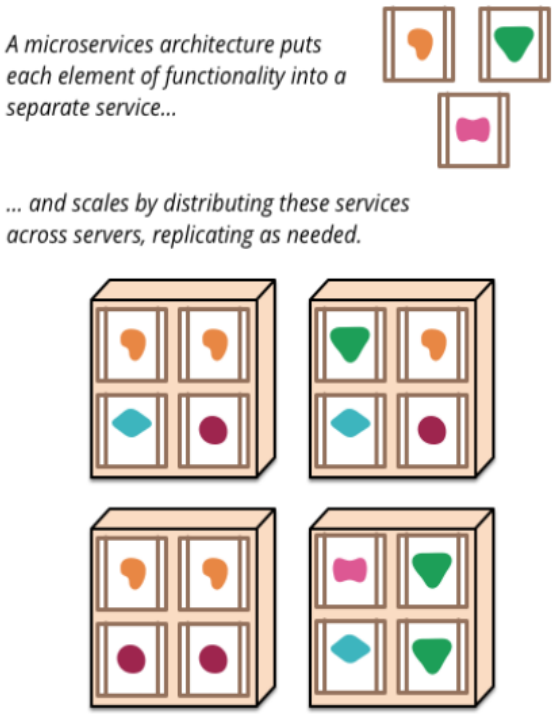


Figure 5 microservices automation

2.3 Hide Implementation Details

This principal reminds us about OOP principle of encapsulation to avoid situations where many services depend on implementation of other service. Because service begins to take too many responsibilities it becomes unstable when the time comes to change some internal implementation of it.

Consider a classical scenario with database.

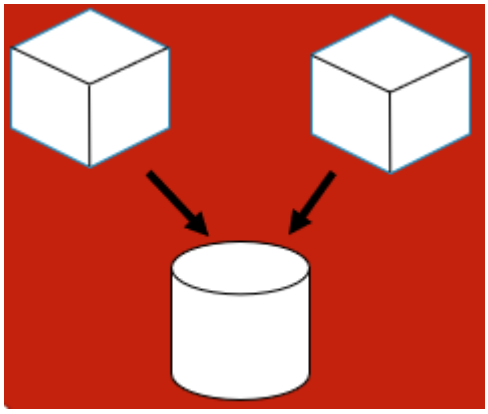


Figure 6 Hiding implementation details

The first thing we note that it is easy to connect two services with one database, it makes sense and it works.
On the next iteration of the development cycle, the team decides to change the database schema. Now both services required to change their implementation details to adopt new schema. This is becoming a problem in Agile development, as we taking twice the time for both services to go through iteration cycle. Both services also most likely contain duplicated code for connecting to the database.

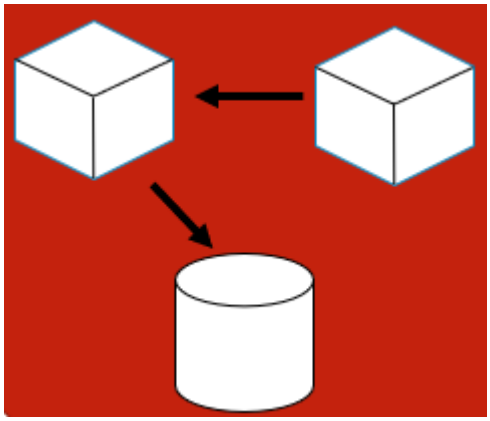


Figure 7 Hiding implementation details

Solution to the problem would be to change request direction of one service from the database to another service. Now only one service is responsible for communication with the database, it encapsulates database models that are mapped to the database schema and no duplicated code is present in other services. Consumers should contact service via HTTP request to receive data according to the communication contract of this service.

2.4 Decentralized Development

Decentralized Development is an ability of development teams to create new services on top of existing infrastructure with minimum communication between other teams and to have permission for innovation [2] when building internal service structure. For this principle to work, it is required to decide on common boundaries between services, what service can accept as a request and what to produce a response. Essentially, cross-team communication is important in designing abstraction for overall application, reducing time spent on deciding how should one service be implemented internally to be able to communicate with others. The term ”Shared Governance” emerging when taking this approach. Development teams gather up together to brainstorm microservices design and pipelines, rather than one architect planning out everything. This also enhances development with fresh ideas from other developers and system design can be evaluated from different perspectives.

2.5 Independent Deployment

One of the most important principals that must be followed when developing microservices. Independent deployment is essentially a goal of microservices architecture. Can we add, change or remove services without breaking other components in a system? If the answer is yes, we are on a right path to success. In development stage, it is inevitable to make mistakes in defining cross-service communication interfaces. It is forgiving thou when taking Agile development approach. It becomes more punishable when moving an application into production stage. If components constantly break other components when changed, added or removed, it should ring a bell that initial design wasn’t right. We should also think if a service should share host machine with another service. It becomes quite expensive to run an individual host machine for every service as an application grows larger. A relatively new technology, called Docker, can help distribute multiple services on one host machine. It isolates applications into containers that does not interfere with the rest of the processes running in the same machine. A point to note, it do not makes sense to run multiple services of the same kind on the same host when replicating. If this host goes down in case of machine error, entire service of an application will go down. Instead, we should consider placing different kinds of service onto one host to share space.

2.6 Consumer First

Design and develop consumer application first. This will clarify what exactly service is for. To understand service functionality, it is important to refer to the business requirements. To help to definE cross service communications, it is recommended to use OpenAPI tools to describe services.

2.7 Isolate Failure

”Microservices aren’t reliable by default” by Sam Newman. Microservices architecture doesn’t make your system more stable, but in fact it increases surface area of failure. When monolithic application is split into separate services, we have to think how to deal with failures for every one of them. It often happens that network encounters problems or a host machine crashes, and service goes down. This shouldn’t stop other services from working. Even worst if services begin to fail slowly, they tight up resources and usually it becomes too late to respond when the whole thing crashes. How to avoid common failures? Start off with minimizing request timeouts, giving it about 2 seconds. This way, if request doesn’t process, we can release a thread on service quickly. Increase amount of thread pools in applications that communicate with many other services. One thread pool per downstream service is sufficient enough to load balance service. If one of the thread pools become exhausted, blocking process of further requests from particular type of a consumer, we insuring that other consumers still can be served. Don’t let services to retry indefinitely. Stop retrying after certain amount of timeouts. This will allow service to either recover or fail faster so we know about it.

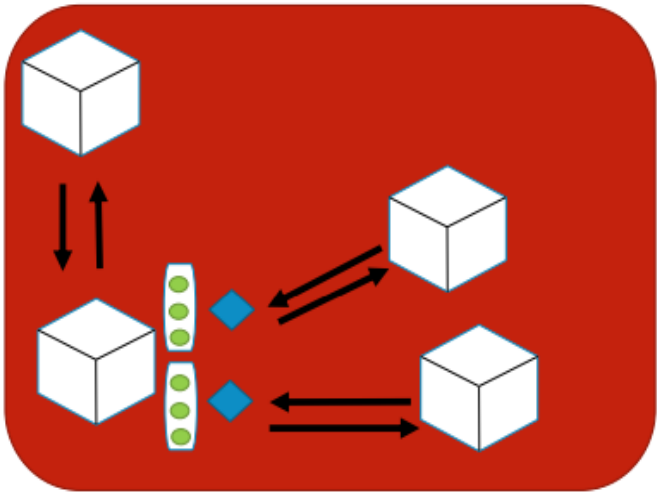


Figure 8 microservices isolate failure

2.8 High Observability

You want to know what is happening in your system. More information we can get about the state of services, better we can respond to situations happening within. It is necessary to introduce a component within a system that is specialized in the management of measurements. [3] Start collecting logs from services and search for abnormal patterns. On large scale applications, it becomes hard to analyze logs from every service, so it is recommended to automate pattern search for unusual behaviour. Collect statistics from services. This can help you to see what service is used more often, so you can replicate it if needed. Statistics also can help to predict peak hours, so you can prepare accordingly. If something fails in microservices, very often it is hidden within the long chain of requests from service to a service. To replicate such failure to fix it maybe be difficult. Correlation id is the technique that can be used here to track down request chains effectively. For every request made, you have to generate a unique id and send it with request down the wire. This way we can follow the chain of request along the way and in case of failure, it becomes easier find and isolate the bug.

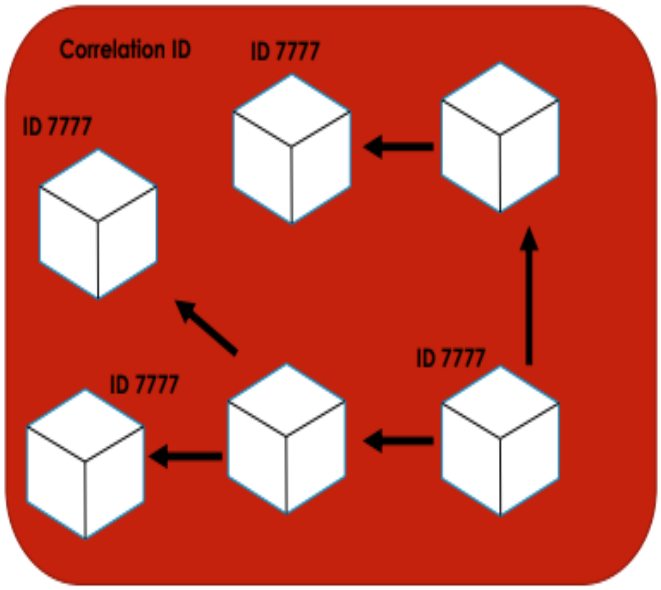


Figure 9 microservices High Observability

2.9 Conclusion

The road to success in microservices comes with price and discipline. One should consider it as a big step forward from monolithic architecture approach. But should you use it or not? It comes to the scope of business requirements. If business requirements are fixed, project scope is within limits of the small to medium project, use monolithic approach to save time and money. If an application is intended to grow with time, it has a lot of functionality or it becomes difficult to maintain the monolithic application, use microservices. It will pay off in time for all the hard work you put into development.

Chapter 3: Methodology

Project development methodology is an important part of developing a complex software and is a basis for managing time and resources. Considering that a subject of dissertation is very new and complicated for us, it was difficult to plan out entire project from the beginning. As a team, we had to pick up bits and peaces, develop idea to show microservices in action, decide on requirements for the project and do quite a big research in this area.

3.1 Requirements gathering

To start the development of microservices somewhere, we had to come up with an idea that is simple enough to show concepts of microservices. That is when "*Micro Service Me Up*" project was contrived. We thought that if user can register, login to our services, be able to manage his/her profile and post some service of their own to a dashboard is good enough to set up good few microservices and hook them up together. Now we had vision of how project emerges at the end of a development process, but the path to success was still unclear.

Requirements

- **User Registration –** User is able to register with the unique email address. Newly registered user is persisted to the authentication database with status of pending user. Password has to be encrypted before saving it to the database. Newly registered user can not access account until email verification process is succeeded.
- **User Activation –** To activate account, verification link is sent to an email address that was specified in registration form. Verification link holds a unique token in form of a path variable. Registration token has an expirity date and set to be valid for 24 hours. Upon visiting verification link, user is informed with verification status, i.e., verified or token is expired.
- **User Authentication –** User is able to login with correct email and password only if registration verification was successful, otherwise access to an account is forbidden. When logged in successfully, unique authentication token is generate and sent back to the client inside a header, and saved as a cookie.
- **Authentication Token –** Authentication token is encrypting user details, i.e., username, authorities, token expiration date and if account is valid. Subsiquently authentication token is included in the header to any secured resources and is validated against in memory token repository.

- **Personal Profile** – When user login for the first time, user profile is created and persisted to the profiles database. Initial user profile contains only email address data with the rest of the parameters being empty. User password is not persisted to the profiles database. It is up to the user to feel out the rest of the profile data through the form from profile page and sent to the profile service to be persisted.
- **Services Dashboard –** Authenticated users have an access to the services dashboard and are allowed to publish and remove their services. Dashboard post consists of title, email address of the user posting it, content body and date. Service publications can viewed individually or as a list of all published services.
- **Email Service –** Email service is a microservice that is responsible for sending system emails like user verification email.

3.2 Planning

Once requirements were set, we had to come up with a rough plan of how we are going to achieve each requirement. We decided to split development into two parts, back-end services development and front-end services development. Back-end development was scheduled for the first semester of the year, starting in mid September and planned to be completed by the end of January. Thanks to the client mocking tools like Postman and Curl, we did not require to develop front end clients until back-end is finished. For the front-end clients we decided to spend roughly 2 months, February and March, of Rapid Application Development. The rest of the time was left for the documentation and preparation of dissertation.

Project Plan

Week 1-4 Initial preparations

- **Technological Research –** commencing research on microservices technologies available on Java platform. This included learning about Spring Cloud, setting up and running starter samples, diving into documentation. We also begin to learn about Docker and how to containerize simple Spring Boot applications. For this purposes we created demo service application. We would like to note that technologies research was an ongoing process and never stopped until the submission date.
- **Architecture Design –** during this phase we picked technological stack we want to use for the project including Spring Cloud Netflix infrastructure for microservices, databases (Neo4j &

MongoDB), cloud host service (AWS) for service deployment and front-end technologies for our client.

- **Project Proposal –** after sufficient research done and architecture design was set in place we were ready to write project proposel and prepare presentation that reflects our intent.
- **Github Repository Setup –** setup Github repository for the project and open Wikis for quick documentation, instructions and problem solutions.

Week 5-6 Spring Cloud Netflix setup

- **Netflix Eureka Discovery Service –** set up discovery service, dockerise service, deploy to AWS Linux instance and test.
- **Netflix Zuul Proxy Service –** set up proxy service, subscribe it to discovery service, dockerise service, deploy to AWS Linux instance and test.
- **Configuring Eureka –** configure discovery subscription renewals and thresholds for optimization.
- **Document Netflix Cloud Technologies –** briefly explaining what is Netflix Cloud and how to use it.
- **Document Docker –** briefly explain what is Docker and how we use it in our project.
- **Demo Service –** setup demo microservice for testing discovery service and proxing to the service.
- **Docker Databases Test Run –** set up dockerized Neo4j and MongoDB databases, deploy to AWS Linux instance and test accessibility.

Week 7 Registration Service

- **Registration Service –** create registration service and connect it to Neo4j database.
- **Neo4j Database –** add sample data to the database to test registration service connection.
- **Spring Cloud Release Integration problem with Neo4j –** solving problems with Maven dependencies as Neo4j libraries are not compatible with currently used Spring Cloud release and its dependencies.
- **Subscribe to Discovery Service –** add configurations to registration service to make it discoverable.
- **Configure Proxy Service –** add routing to registration service for proxying.
- **Test Proxy –** testing request proxying to registration service to query database.
- **Documentation –** documenting encountered problems and solutions.

Week 8-9 User Verification and Activation

- **Spring Cloud Release Migration –** changing project Spring Cloud dependencies for another release to solve compatibility problems with Neo4j libraries.
- **Verification Token –** create a token upon registration and persist to database.
- **User-Token Relationship –** create relationship between user and token.
- **Verification end-point –** create API end-point that verifies registration token. Perform check against database tokens and handle scenarios where token exist and is valid, token does not exist and token exists, but is expired.
- **Unique Users –** handle duplicated user registration.
- **Email Service** – create an email service to handle system emailing, particularly to send email verification link.
- **Email Service API –** create an email service API to be used by other services to make requests to Email Service.
- **AWS and Docker Integration Problems –** searching for solution to solve problem with deploying docker container to AWS Linux instances. Container exposing private IP address of an instance rather than public therefore requests are not getting through.

Week 10-12 Microservices Authentication

- **Research –** research authentication methods for microservices architecture.
- **Custom Authentication Tokens –** attempt to implement custom authentication tokens and integrating it into microservices ecosystem. (**Failed**)
- **Researching OAuth2 –** investigating OAuth2 technology.
- **Researching Spring Cloud Security –** running Spring Cloud security samples to learn how it works.
- **Authentication Service** – adapting Spring Cloud security samples into Authentication service.
- **Connecting to Database –** implementing Neo4j database interface to Authentication service to validate user accounts against existing data.
- **Testing Authentication from Database –** testing login with registered and verified users from Neo4j database.
- **Examining Authentication Token –** learning about JWT token in OAuth2 context.
- **Dockerising Authentication Service –** creating docker image and container for authentication service.
- **AWS deployment –** deploying authentication service to AWS Linux instance.
- **Testing Authentication from Cloud –** testing authentication service behaviour after deploying it on Cloud, subscribing it to discovery service and routing with proxy service.

Week 13 Profile Service

- **Profile Service –** design and develop service.
- **Connect to MongoDB –** add service class, repository interface and model to persist to MongoDB.
- **Test locally –** test service locally with client mocking tools.
- **Add to Cloud –** subscribe with Discovery and add routing to Proxy.
- **Dockerise –** create an image and container for service.
- **AWS Linux –** add service to the Linux instance on AWS.
- **Test on Cloud –** test integration of service with microservices eco system on cloud.

Week 14 Dashboard Service

- **Dashboard Service –** design and develop service.
- **Connect to MongoDB –** add service class, repository interface and model to persist to MongoDB.
- **Test locally –** test service locally with client mocking tools.
- **Dockerise –** create an image and container for service.
- **AWS Linux –** add service to the Linux instance on AWS.
- **Test on Cloud –** test integration of service with microservices eco system on cloud.

Week 15-16 Clients Technologies Research

- **Refactoring Spring Cloud Authentication Sample –** clean up Authentication sample to suit our development practices and principals.
- **AngularJS Client –** examining AngularJS client sample that come together with Spring Cloud OAuth2 Authentication service. Learning how to program with AngularJS. Refactoring code to make it more clear.
- **ReactJS Client –** deploying simple React application as a Spring Boot service. Integrating with other frameworks like jQuery and Bootstrap CSS/JS. Researching Webpack as a solution to integration.

Week 17-18 Rapid Client Development

- **AngularJS Client –** adapting Angular client to make requests to microservices and consume responses. Developing client to target mobile devices. Dockerising as a Spring Boot application.

- **Python Flask Microservice –** adding the Flask API written in python to serve and handle registration page. Demonstrating the freedom of technologies that can be used when developing Microservices architecture, as well as, simulating OpenID services of our own.
- **ReactJS Client –** searching for solutions to the problems arised with using Bootstrap classes. Classes are not applied to html elements.

Week 19 Rapid Client Development Cont.

- **AngularJS Client –** finishing off microservices network communications. Styling application and adding adaptivity to mobile devices.
- **ReactJS Client –** removing client from Spring Boot application and launching Node.js server to host it. Implementing alternative solution to inject Bootstrap framework via open source React components. Removing Webpack configurations as no longer required. Learning how to use React Routing component.
- **Fixing Registration Responses –** sending plain text response instead of json. Recapping other services for similar problem.
- **AngularJS client –** Reachering how ng-model, ng-controller works together. Design mobile clirnt html pages (UI stuff).
- **Python Flask Microservice –** researching for solving why cannot access the server using network (except localhost).

Week 20 Rapid Client Development Cont.

- **AngularJS Client –** creating docker image and deploying container to AWS Linux instance.
- **ReactJS Client –** coding user registration component and testing against cloud microservices system. Looking for solution to allow Cross Origin Requests (CORS) as it became impossible to test registration component that is making a requests to microservices that are deployed in AWS Linux instances in USA.
- **Configuring Proxy for CORS –** adding a copy of Proxy service to cloud that is handling CORS requests with extra configurations.
- **Investigating OPTIONS HTTP Method –** researching ES6 standard and how browsers sending requests with OPTIONS method before fetch function makes actual API call.
- **OPTIONS Request Handling in Proxy –** solving problems with dispatching response to an options request to Proxy and further forwarding of actual request to the microservices.
- **Cutting off Development** – stop project development by the end of this week.

Week 21 Dissertation Preparation

- **Dissertation Report Structure** – defining structure of the document.

- **Dissertation Introduction** – introduce to our project idea, intent of the research, describe aims and objectives.
- **Microservices Development Principles** – brief introduction into microservices, what it is all about and rules of thumb to follow during development.
- **Technologies review** – write about technologies used during development and give a brief instructions on how to start.
- **System design** – write about technique details used during development and explan how the compnents working together.

Week 22 Dissertation Preparation Cont.

- **Methodologies –** write about importance of using methodologies processes during development of complex peace of software. Feature a project plan and development storyboard. How we went about organising development processes and allocation of responsibilities. What research was undertaken and what methodology was used throughout development.
- **System design –** feature our project software that was developed, describing architecture for microservices and clients, revieling database schemas. Illustration of the System design via UMLs and images.
- **System evaluation –** write about our achievements in project, what problems encountered during development, what was solved and left unsolved.
- **Conclusion –** conclude our research in the world of microservices, tell about our feelings and impressions, add recommendations for future development.

3.3 Organisation

At the beginning of the project development cycle we organised our team meeting and working schedule to dedicate our time to the Microservice Me Up project. The idea to research and develop Microservices Architectures came up very quickly during our discussions about what modern IT industry look like this days, which direction they looking. We also had an experience in developing Web application back-end with multiple different clients communicating with HTTP and RESTful approach. Why not to try create a whole sworm of them?

We contacted our project supervisor, Dr. Brian McGinley during the first week of commencement to the project, to discuss our weekly meeting plan, inform him about our idea and plan for the first week. After college timetable was settled for the semester, we agreed with Dr. Brian to meet on Monday every week. For us it was a valuable time to receive guidance from him as to how to commence with the research of

such a big and complicated topic. During the Monday meetings with supervisor, we had a review of our weekly work, informing about research progress in new for us web technologies, ensuring that the work scope for the following week is set correctly.

Dr. Brian recommended us to create a project plan and schedule development for the software development process. It was difficult to plan too far ahead as we were not sure how long it might take to set up working environment and try out new technologies. We tried to develop an abstract idea of how end product might look like and adjust our schedule plan. While we work on essential components of the project, we could still see far enough ahead to make a choice in work flow direction.

As a part of an organisational process we also had to set up quite big development environment that included installation of new software on local and remote operating systems, management of the AWS accounts to create and configure Linux instances, setting up connecting interfaces locally to access and operate remote instances from shell, creating and organising GitHub repository for versioning control and collaborative work, utilizing DockerHub Organization option to share software images between our own user accounts.

3.4 Research

Brief list of our researching:

- Microservices Architecture – reading materials on concepts behind microservices, how architecture is design and function, what technologies are commonly used for software realisation.
- Spring Cloud Netflix technology – study and learn how to use a set of libraries to create essential infrastructure for microservices and wire up everything together. It was required to complete some tutorials online and run different samples to get a grasp of it.
- Docker software – was complete a new concept for us. We had to dig into documentation to understand how to create necessary files and execute commands. We decided to learn it as soon as possible and used it extensively throughout development.
- JWT tokens – another concept that we never looked at before as an alternative to a traditional session based security. Concepts of JWT tokens were not too hard to understand, but making it work together with all microservices was difficult task. We had to go through many examples and approaches to get a grasp of it.

3.5 Methodology

As a students we never had a chance to work in real environment where methodologies are used as basis to development of software. In theory we learned about various methodologies, it's processes and techniques, but without practical usage and sufficient resources we can not say we used any particular one in its full form. We were picking techniques that are reasanble enough to enchance our project development and save time where we can.

While examining our project development lifecycle, we can describe our patter as an iterative development approach that is used in Agile environments. Similar to the Scrum technique, we pick a next goal and sprint to develop working version, white box testing it, examining problems, if possible fixing them along the way and retesting.
Another iteration begins when integrating new microservices into the cloud eco system. It has a tendency to introduce completely different and unexcpected problems that can not be revieled in local development environment. This brought us difficulties into developing in different environments and test them separately. In industries it is common to have cloud development and testing environments set in place, as for us we had to deal with it as it comes at us, often preparing solution locally and deploying it to cloud for testing. Overall, system was growing incrementaly and was showing satisfacting degree of stability.

Not every aspect of methodology processes and software components received full attention, and had to be sacrificed for the abviouse reasons, the lack of resources like time and manpower. Requirements were also flixbile enough to be changed if were running out of time, because project did not have a particular customer, although our supervisor was also a stackeholder and adjusted requirements to make sure we stay in reasanble scope of dissertation requirements to be achieved by honors students in software development.

During the times when difficult problems encountered and overall slowing down development process, we had to use pair programming (XP) approach to respond it. Together it was faster to examine the problem, locate the root cause and find suitable solution.

During our project lifecycle we saw the benefits of applying various methodologies techniques. Knowledge of methodologies in theory came handy into practice when we begin to see this patterns in action. We believe it improved our development for this dissertation and produced positive learning outcomes from firsthand experience.

3.6 Github

Github API was our main code collaborative development tool to increment the progression in development of software for Micro Service Me Up project. Standard collaborative practices in using Github were applied throughout development, i.e., make a pull request before committing to master branch to avoid conflicts, usage of wikis to document problems, describe the usage of new technologies and maintain project plan. We would like to note that it is handy to collaborate on development of microservices, because each service is contained in its own directory as a stand alone project and can be changed by developers without major conflicts with other services in development by other people. Github repository link: https://github.com/TanqEdPublic/Micro-Service-Me-Up

Chapter 4: Technology Review

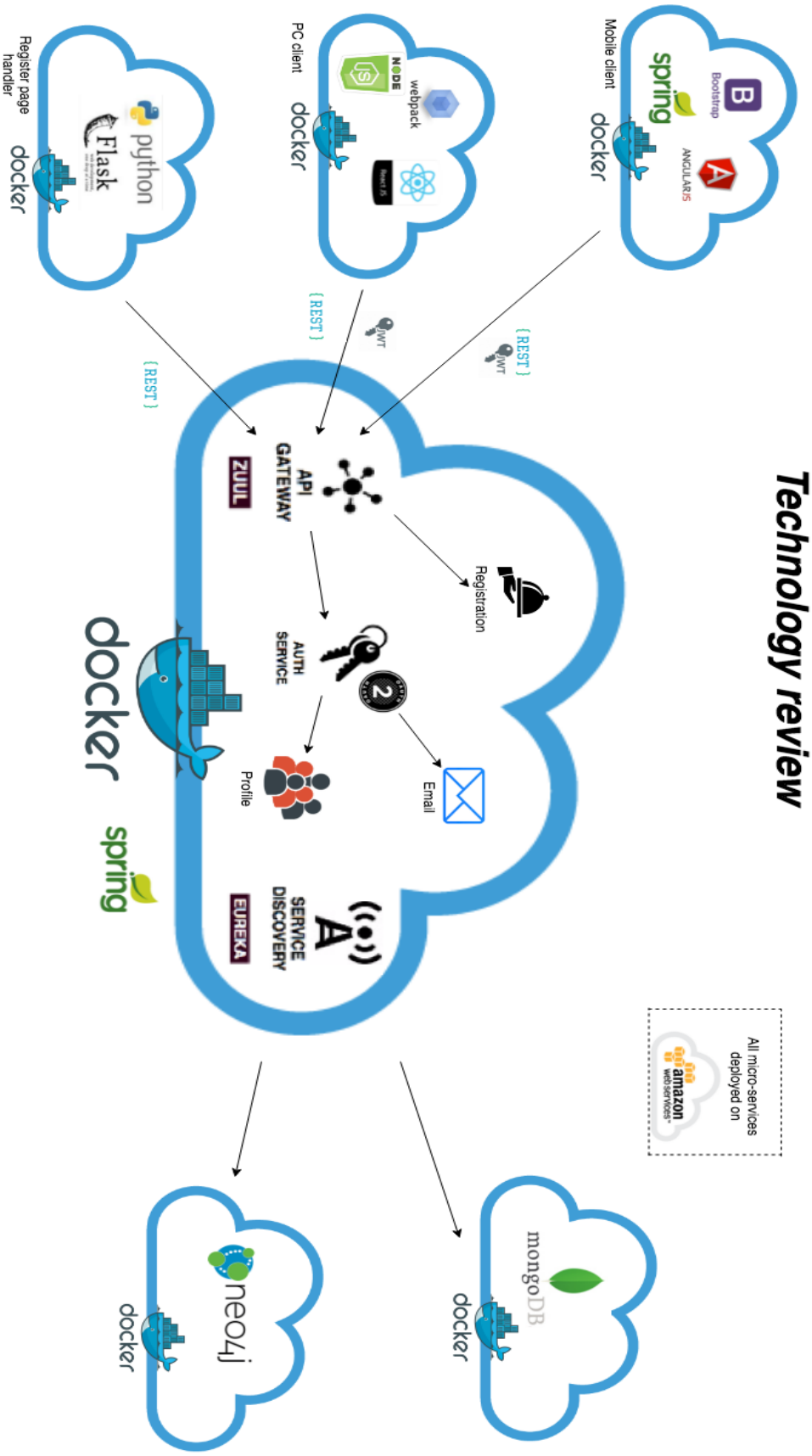


Figure 10 Technology Review

4.1 Server Side Technologies

4.1.1 Maven & Spring Boot

Apache Maven is a software project management and comprehension tool. It is a tool that can now be used for building and managing any Java-based project. We hope that we have created something that will make the day-to-day work of Java developers easier and generally help with the comprehension of any Java-based project.

Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal there are several areas of concern that Maven attempts to deal with making the build process easy, providing a uniform build system, providing quality project information, providing guidelines for best practices development and allowing transparent migration to new features. Based on the concept of a project object model (POM), *Maven* can manage a project's build, reporting and documentation from a central piece of information.

There is similar project build tool called ***Gradle***. *Maven* uses an XML-based configuration and *Gradle* uses the configuration of a domain-specific language Groovy. Since our course is linked to maven, so we use maven in this *Micro Service Me Up* project.

Takes an opinionated view of building production-ready *Spring* applications. ***Spring Boo*t** favors convention over configuration and is designed to get you up and running as quickly as possible.

Spring Boot allows us to quickly create a Spring-based project, and let this Spring project run we only need a little configuration can be:

- Stand-alone Spring applications
  Spring Boot can run as a jar package. Very convenient to run a *Spring Boot* project, we only need to run the *java -jar xx.jar class*.
- Embed Tomcat, Jetty or Undertow directly
  No need to deploy WAR files.
- Provide opinionated 'starter' POMs to simplify your Maven configuration
  Using *Spring* or *Spring MVC* we need to add a large number of dependencies, and many of these dependencies are fixed, where *Spring Boot* can help us simplify *Maven* configuration through the starter.
- Automatically configure Spring whenever possible

- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

Quick Start

Code samples (*maven* + *spring boot*) :

`Pom.xml`

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

*NOTE: Different dependency have multi version, it is necessary to check the version you choose is/not correct!!!

`hello/SampleController.java`

```
package hello;
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;
@Controller
@EnableAutoConfiguration
public class SampleController {
    @RequestMapping("/")
```

```
    @ResponseBody
    String home() {
        return "Hello World!";
    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(SampleController.class, args);
    }
}
```

4.1.2 Spring Cloud Netfilx

Spring Cloud is a set of frameworks for implementing microservices based on ***Spring Boot***. It provides components such as configuration management, service discovery, circuit breakers, intelligent routing, micro-agents, control buses, global locks, decision campaigns, distributed sessions, and cluster state management that are required for the development of microservices (introduced above). The most important thing is that using it with the spring boot framework will make it very easy for you to develop cloud services for the microservice architecture.

Spring Cloud Netflix [4] is a set of sub-frameworks of *Spring Cloud*, developed by Netflix and later incorporated into the *Spring Cloud* family. Its main components:

- ***Eureka***, service registration and discovery, it provides a service registry, service discovery client, and a convenient interface to view all registered services. All services use Eureka's service discovery client to register themselves with Eureka's servers.
- ***Zuul***, gateway, all client requests to access back-end services through this gateway. He can use a certain routing configuration to determine which service a particular URL handles. And get the registered service from Eureka to forward the request.
- *Ribbon*, load balancing, when a Zuul gateway sends a request to an application of a service, if a service starts multiple instances, it will be sent to a service instance through a certain load-balancing policy through the *Ribbon*.
- *Feign*, for service clients, if they need to access each other, can use the RestTemplate or Feign client access. It uses *Ribbon* for load balancing by default
- *Hystrix*, monitoring and circuit breakers. We only need to add the Hystrix tag to the service interface to implement monitoring and circuit breaker functions for this interface.

In the server side of *Micro Service Me Up* project, we implement basic components: Eureka and Zuul. Eureka discovers all microservices (authentication service, email service, registration service & profile service) and gives each service an ID. It enables the microservice can connect others just using the ID because Eureka knows the actual IPs. Zuul acts as the gateway. It directs all client requests to our authentication service, which achieves clients need a valid identity to access our services.

We use *Spring Cloud Netflix* with *Spring Boot*. If you want to use it in your original Java Web application, you can also practice it by adding relevant configuration.

Quick Start

Code samples (Development for Registration and Monitoring Center – *Eureka*)

```
// ignore import
@SpringBootApplication
@EnableEurekaServer
@EnableHystrixDashboard
public class EurekaServerApplication {
public static void main (String[] args) {
    SpringApplication.run(EurekaServerApplication.class, args);
}
}
```

The **@SpringBootApplication** tag here indicates that the current application is a *Spring Boot* application. So you can use the main function directly in the IDE to start the application, you can also use the command line to start the package. And, you can also start the packaged war package with a server like Tomcat.

Use the tag **@EnableEurekaServer** to start the components of the Eureka service registry during startup. It will listen on a port, the default is 8761, to receive service registration. And provide a Web page, open, you can see the registered service.

Adding **@EnableHystrixDashboard** will provide a monitoring page where we can enter the address of the service to be monitored, and we can view the call of the interface enabled with Hystrix monitoring.

*NOTE: In order to use the above components, we need to add corresponding dependencies in the Maven POM file, such as using *spring-boot-starter-parent*, relying on *spring-cloud-starter-eureka-server* and *spring-cloud-starter-hystrix-dashboard* etc.

Then configuring the properties in *YAML* format; so an **application.yml** will be in configuration file:

```
server:
  port: 8761
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

Code samples (gateway – *Zuul*)

```
@SpringBootApplication
@EnableZuulProxy
public class GatewayServiceZuulApplication{
public static void main (String[] args) {
    SpringApplication.run(GatewayServiceZuulApplication.class, args);
       }
}
```

Used **@EnableZuulProxy** to handle passing contexts between filters, such as: where the request is routed, errors, HttpServletRequest, HttpServletResponse, data is kept in each request's ThreadLocal.

application.yml configurations:

```
zuul:
routes:
   demo:
      path: /demo/**
      serviceId: DEMOMICROSERVER
```

Which means the client server DEMOMICROSERVER can be accessed via the ip of zuul.
e.g. http://zuul-ip/demo/**
Also need to add *spring-cloud-starter-zuul* dependency in the Maven POM file.

Code samples (microservice – *Eureka Client*)

```
// ignore import
@SpringBootApplication
@EnableEurekaClient
@RestController
public class EurekaClientApplication{
public static void main (String[] args) {
    SpringApplication.run(EurekaClientApplication.class, args);
}
@RequestMapping("/greeting")
    public String greeting() {
        return "Hello from EurekaClient!";
    }
}
```

@EnableEurekaClient here enables *Eureka* is able to discovery such client.

@RequestMapping acts as the normal router function as showed in *spring boot* project.

application.yml configurations:

```
spring:
  application:
    name: spring-cloud-eureka-client
server:
  port: 0
eureka:
  client:
    serviceUrl:
      defaultZone: ${EUREKA_URI:http://localhost:8761/eureka}
  instance:
    preferIpAddress: true
```

Also need to add *spring-cloud-starter-eureka and spring-boot-starter-web* dependencies in the Maven POM file.

4.2 Client Side Technologies

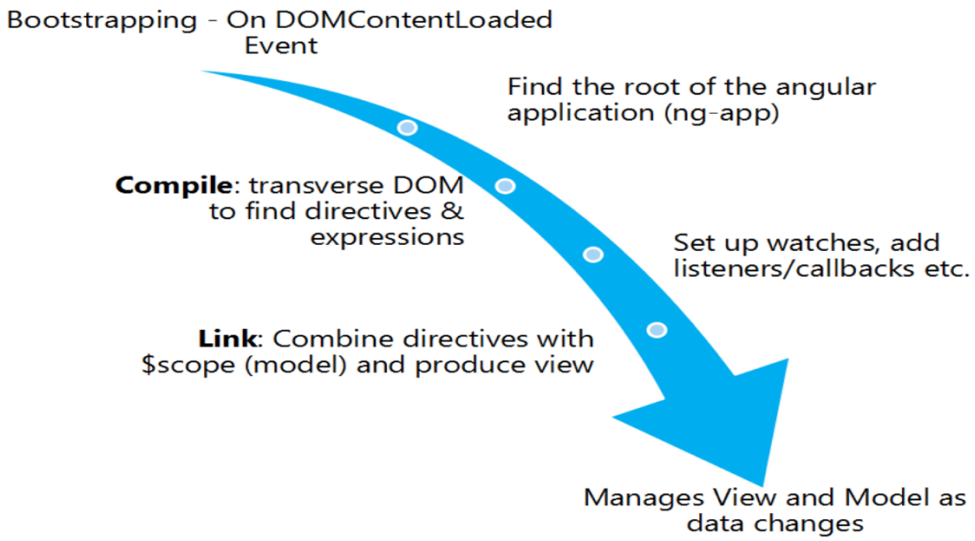
4.2.1 Mobile-client (*Spring with OAth2, Bootstrap & AngularJS*)

The mobile client handles a web client, designing for mobile devices. The mobile web app route is faster and cheaper than the native mobile app route, especially when the objective is to support a wide range of devices. Considering cross-platform application for the mobile client, we pick web application. The simple reason is that we do not want to bring troubles to the client for downloading and installing the app from google play store, apple store etc. On the other hand, developing different mobile applications needs different technologies such as *Android SDK*, *Objective-C* for ios, and *c#* for windows app. Actually, the main point for the *Micro Service Me Up* project is microservice. We add clients just to show how micriservices in the project working together, and we do not require capabilities native to the mobile device (such as the movement sensor and so on) currently.

- ***Spring* framework with *OAuth2***

  For the mobile client web page handling, we choose *Spring* again. The main reason is that we implement ***OAuth2*** technique in our authentication server which uses spring framework. In order to achieve authentication service, the client side needs config the ***OAuth2*** and pair to the server one. Actually, most modern techniques can work with the ***OAuth2***, but to the limit time for us, we only found the way through spring by now.

- ***Bootstrap & AngularJS***



[5]

Figure 11 Bootstrap structure

Bootstrap is an open source toolkit for developing with HTML, CSS, and JS. Quickly prototype your ideas or build your entire app with our Sass variables and mixins, responsive grid system, extensive prebuilt components, and powerful plugins built on jQuery.

HTML is great for declaring static documents, but it falters when we try to use it for declaring dynamic views in web-applications. ***AngularJS*** lets you extend HTML vocabulary for your application. The resulting environment is extraordinarily expressive, readable, and quick to develop.

With ***AngularJS***, manipulating data isn't a grab and inject sort of affair. Things are data-bound so we don't need to do all that barbaric grab and inject stuff. We should be able to bind a variable to each component (button or collapse) and then toggle it based on the true/false nature of that variable. This is why we cannot just use ***Bootstrap*** JavaScript. It relies on ***jQuery*** and we don't want ***jQuery*** rummaging around our Angular projects. If we try to bind variables to the components, it won't work.

Quick Start

Code samples for *Bootstrap*: https://getbootstrap.com/docs/4.0/getting-started/introduction/

Code samples for *AngularJS* (data binding)

```
<div ng-app="">
  <p> Name : <input type="text" ng-model="name"></p>
  <h1>Hello {{name}}</h1>
</div>
```

Code samples for *AngularJS* (ng-model)

```
<div ng-app="myApp" ng-controller="myCtrl">
      <input ng-model="name">
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
```

```
    $scope.name = "John Doe";
});
</script>
```

For more functionalities such as controller, scopr, filter etc. you can learn from : https://www.w3schools.com/angular/default.asp

Code samples for *Bootstrap* & *AngularJS* in *Spring*

In *spring,* there is a tool called ***WebJars***, the client-side web libraries (e.g. jQuery & Bootstrap) packaged into *JAR* (Java Archive) files. For enabling *Bootstrap* & *AngularJS* in *Spring Framework*, just by adding related dependencies to the POM.xml file:

```
<dependency>
    <groupId> org.webjars </groupId>
    <artifactId> angularjs </artifactId>
</dependency>
<dependency>
    <groupId> org.webjars </groupId>
    <artifactId> bootstrap </artifactId>
</dependency>
```

4.2.2 PC-client (*NodeJS & ReactJS*)

To show wider range of possibilities to develop clients for micro services, we decided to showcase NodeJS runtime and ReactJS library. Client is ment to be used by pc users in browsers as it is currently not supporting screen adaptivity for smaller devices. For this case mobile client is developed.

- ***NodeJS***

  NodeJS is an open source community project, a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

  It is used as a base for ReactJS client and host for static resources like *index.html* and bundeled javascript files.

Download NodeJS: https://nodejs.org/en/

- ***ReactJS***

  ReactJS library is a JavaScript library for building interactive user interfaces. You can design simple views for each part of your application, and React will efficiently update and render just the right components when data is changed.

  Components

  ReactJS is not using html templates to declare UI, instead developer have to create components in JavaScript. Each component can manage its own state and lifecycle, and then integrated into complex UIs of many components.

  There are 2 types of components that are general rule of thumb to follow:

  - **Presentational Components** – are concerned with how application look. Contains DOM markup and styles. Data and callbacks are received via props. They do not specify how data is loaded or changed. Most of the time presentational components are stateless and do not maintain state unless required for UI state rather than data. Generally written as functional components unless they need state.
  - **Container Components** – are concerned with how application works. Usually don't have any DOM markup except for some wrapping divs, and never have any styles. Provide data and behaviour for presentational components. Are often stateful and serve as a data source.

  JSX Syntax

  To inject DOM markup from JavaScript, React uses JSX syntax. This allows you to write html tag blocks inside JavaScript file. This blocks are mapped to virtual React DOM and then transformed into actual DOM.
  Instead of artificially separating technologies by putting markup and logic in separate files, React separates concerns with loosely coupled units called "components" that contain both.
  It is also possible to embed any JavaScript expression in JSX by wrapping it in curly braces.
  For example:

```
var today = new Date();
const element = <h1>Hello, world! It's {today}.</h1>
```

To quickly start with ReactJS, follow simple instructions from:
https://reactjs.org/tutorial/tutorial.html

4.3 Register page handler (*python flask*)

The purpose of this part is to simulating how *OpenID* works.

OpenID allows you to use an existing account to sign in to multiple websites, without needing to create new passwords. You may choose to associate information with your OpenID that can be shared with the websites you visit, such as a name or email address. With OpenID, you control how much of that information is shared with the websites you visit, your password is only given to your identity provider, and that provider then confirms your identity to the websites you visit. Other than your provider, no website ever sees your password, so you don't need to worry about an unscrupulous or insecure website compromising your identity.

This register page handler is a single web page application, only provide account creation for the ***Tanqed*** platform. Then the user can use this account to request services through other microservices. Similar to Google and Facebook account, which can be used to login mang other apps or websites.

Flask is a popular web framework that is implemented in ***Python***. "*Micro*" framework is the notable feature of Flask. It means that Flask aims to keep the core simple, but at the same time it is easy to expand. By default, Flask doesn't include database abstraction layers, form validation, or anything else that already has multiple libraries that it can handle. However, Flask supports using extensions to add these features to applications. Numerous extensions provide database integration, form validation, upload processing, and various open authentication technologies. These Flask features make it very popular in Web development.

Quick Start

A minimal application looks like this:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
```

```
    return 'Hello World!'
if __name__ == '__main__':
    app.run()
```

First we imported the class ***Flask***. The instantiation of this class will be our application. The first parameter is the name of the application module. If you are using a single module (as in this example), the first parameter should use __name__. Because it depends on if it is started as a separate application or imported as a module, the name will be different ('__main__' corresponds to the actual imported name). Next, we create an instance of this class. We pass it the name of the module or package. By this way Flask will know where to look for templates, static files, etc. Decorator ***route()*** is used to tell Flask which URL to fire our function. Finally we start the local server with our function ***run()*** to run our application. *If __name__ == '__main__*': Make sure that the server will only run when the script is executed directly by the Python interpreter, not when imported as a module.

Save it as hello.py (or a similar file) and run it with the Python interpreter. Make sure your application is not called flask.py, as this will conflict with Flask itself.

```
$ python hello.py
 * Running on http://127.0.0.1:5000/
```

Now visit http://127.0.0.1:5000/ and you will see your “Hello World” greeting.

4.4 Databases

For storing data, we use ***Neo4j*** and ***MongoDB*** database. In the past several years, we practiced much relational database like *MySQL*. *MySQL* is good, but we found it is more suitable to using *Neo4j* and *MongoDB* in the *Micro Service Me Up* project after research. On the one hand, we would like to try new things. On the other hand,

Both of them are ***NoSQL*** database. *NoSQL* (originally referring to "non SQL" or "non relational") is a generic term for database management systems that are different from traditional relational databases. There are many significant differences between the two, the most important of which is that NoSQL does not use SQL as a query language. Its data storage may not require a fixed tabular schema, and often avoids the use of SQL JOIN operations, which generally have horizontal scalability features. A ***NoSQL*** database provides a mechanism for storage and retrieval of data that is modelled in means other than the tabular relations used in relational databases. Sometimes the data structures used by *NoSQL* databases are also viewed as "more flexible" than relational database tables.

4.4.1 Neo4j

Neo4j is a graph based database management system by Neo Technology, Inc. *Neo4j* is implemented in Java and accessible from software written in other languages using the Cypher Query Language through a transactional HTTP endpoint, or through the binary 'bolt' protocol.

In *Neo4j*, everything is stored in the form of either an edge(Relationship), a node, or an attribute. Each node and edge can have any number of attributes. Both the nodes and edges can be labelled. Labels can be used to narrow searches.

For example: [6]

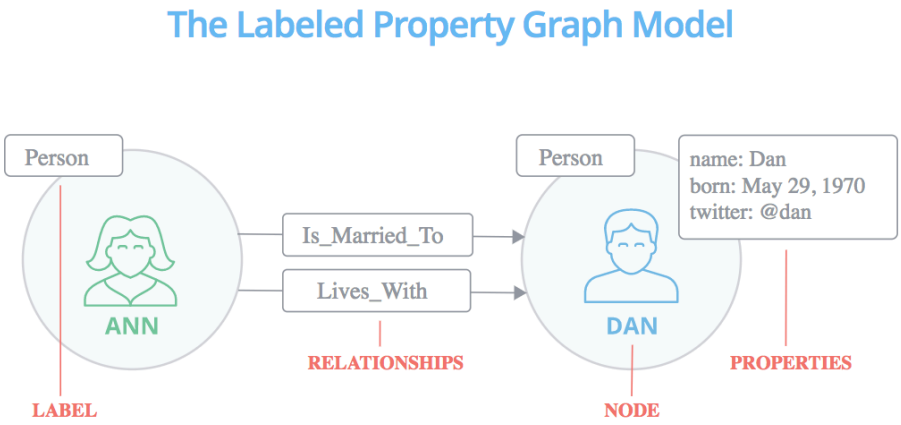


Figure 12 Neo4j basic structure

Quick Start

Visiting *Neo4j* in *Spring Boot* can be easily implemented with *Spring Data Neo4j*:

Add *neo4j* dependency in the Maven POM file:

```
<dependency>
    <groupId> org.springframework.boot </groupId>
    <artifactId> spring-boot-starter-data-neo4j</artifactId>
</dependency>
```

Use annotations to create graph:

```
@NodeEntity
public class User {
```

```
    @GraphId
    private Long id;
    private String name;
    @Relationship(type = "is", direction = Relationship.OUTGOING)
    private Role role;
```

Implementing org.springframework.data.neo4j.repository.GraphRepository

```
@Repository
public interface UserRepository extends GraphRepository<User>{
    extra method ... ...
}
```

Similar to *JPA* (Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation) to *MySQL*.

4.4.2 MongoDB

MongoDB is an open-source document database and leading NoSQL database. It is written in *C++*. Actually, *MongoDB* is a product between a relational database and a non-relational database. It is the most feature-rich and relational database among non-relational databases.

MongoDB is designed to provide scalable, high-performance data storage solutions for web applications. *MongoDB* stores data as a document, and the data structure consists of key=>value pairs. *MongoDB* documents are similar to JSON objects. Field values can contain other documents, arrays, and document arrays.

For example:

[7]

Figure 13 Mongo data sample

Figure 4.3 Mongo data sample

Quick Start

Visiting *MongoDB* in *Spring Boot* can be easily implemented with *Spring Data Neo4j*:

Add *mongoDB* dependency in the Maven POM file:

```
<dependency>
    <groupId> org.springframework.boot </groupId>
    <artifactId> spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

Use annotations to create graph:

```
public class Item {
   @Id
   private Long id;
   private String title;
   private Role content;
   ... ...
```

Implementing org.springframework.data.neo4j.repository.GraphRepository

```
@Repository
public interface ItemRepository extends MongoRepository<Item, String>{
    extra method ... ...
```

```
}
```

Similar to *JPA* (Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation) to *MySQL*.

4.5 Docker

Docker is the world's leading software container platform. Developers use Docker to eliminate "works on my machine" problems when collaborating on code with co-workers. Operators use Docker to run and manage apps side-by-side in isolated containers to get better compute density. Enterprises use Docker to build agile software delivery pipelines to ship new features faster, more securely and with confidence for both Linux, Windows Server, and Linux-on-mainframe apps.

Docker automates the repetitive tasks of setting up and configuring development environments so that developers can focus on what matters: building great software. Developers using Docker don't have to install and configure complex databases nor worry about switching between incompatible language toolchain versions. When an app is dockerized, that complexity is pushed into containers that are easily built, shared and run. Onboarding a co-worker to a new codebase no longer means hours spent installing software and explaining setup procedures. Code that ships with Dockerfiles is simpler to work on: Dependencies are pulled as neatly packaged Docker images and anyone with Docker and an editor installed can build and debug the app in minutes.

Docker is usually used in the following scenario:

- Web application automation package and release;
- Automated testing and continuous integration, release;
- Deploy and adjust databases or other back-end applications in a service-oriented environment;
- Build or extend your existing OpenShift or Cloud Foundry platform from scratch to build your own PaaS environment.

In the *Micro Service Me Up* project, every parts (includes clients, microserices in server side and the databases) are packaged with docker. That makes quite easy to deploy components respectively.

Quick Start

Simple commands for Docker:

- Show images

  `$ docker images or $ docker images –a`

- Delete a image

  `$ docker rmi IMAGE_ID/NAME`

- Show Containers

  `$ docker ps or $ docker ps –a`

- Operate containers

  `$ docker start/stop/restart/rm CONTAINER_ID/NAME`

Code sample for dockerizing and deploying a Maven, Spring Boot project:

Containerize an existing project: Docker has a simple Dockerfile file format that it uses to specify the "layers" of an image. So let's create src/main/docker folder and create a Dockerfile in Spring Boot project:

```
FROM frolvlad/alpine-oraclejdk8:slim
VOLUME /tmp
ADD demo-0.0.1-SNAPSHOT.jar app.jar
RUN sh -c 'touch /app.jar'
ENV JAVA_OPTS=""
ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ]
```

demo-0.0.1-SNAPSHOT.jar can be founnd in the **target** folder, change the file name with your .jar file. you can also use IDE (Eclipse/Intellij etc.) to build a .jar file.

Add a new plugin to **pom.xml** :

```
<properties>
   <docker.image.prefix>ImageName</docker.image.prefix>
</properties>
<build>
    <plugins>
```

```
        <plugin>
            <groupId>com.spotify</groupId>
            <artifactId>docker-maven-plugin</artifactId>
            <version>0.4.11</version>
            <configuration>
                <imageName>${docker.image.prefix}/${project.artifactId}</imageName>
                <dockerDirectory>src/main/docker</dockerDirectory>
                <resources>
                    <resource>
                        <targetPath>/</targetPath>
                        <directory>${project.build.directory}</directory>
                        <include>${project.build.finalName}.jar</include>
                    </resource>
                </resources>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Create Docker image

- cmd to the prohect folder
- run `$ mvn package -Dmaven.test.skip=true docker:build`

*Note: Make sure Maven is installed.

Run Docker image:

```
$ docker run --name=ProjectName -p 8080:8080 -t ImageName
```

4.6 Amazon Web Services (AWS)

AWS is a cloud based platform that provides many services like development tools (Lambda), database management systems, testing tools and virtual environment for production hosting. For our project we use Linux micro-instances to host micro services from cloud. Linux is a simple OS that requires few configurations, mainly opening connection ports to communicate with it from outside. Thanks to a Docker, it was required even less dependency installations to support microservices.

Chapter 5: System Design

Goal:

- Creating our own spring cloud structure.

- Servering two cross-platform clients (pc & mobile).
- Simulating OpenID, provide our own platform accounts. With the account accessing our microservices.
- Dockerize each microservice, deploying to AWS.

5.1 Back-end

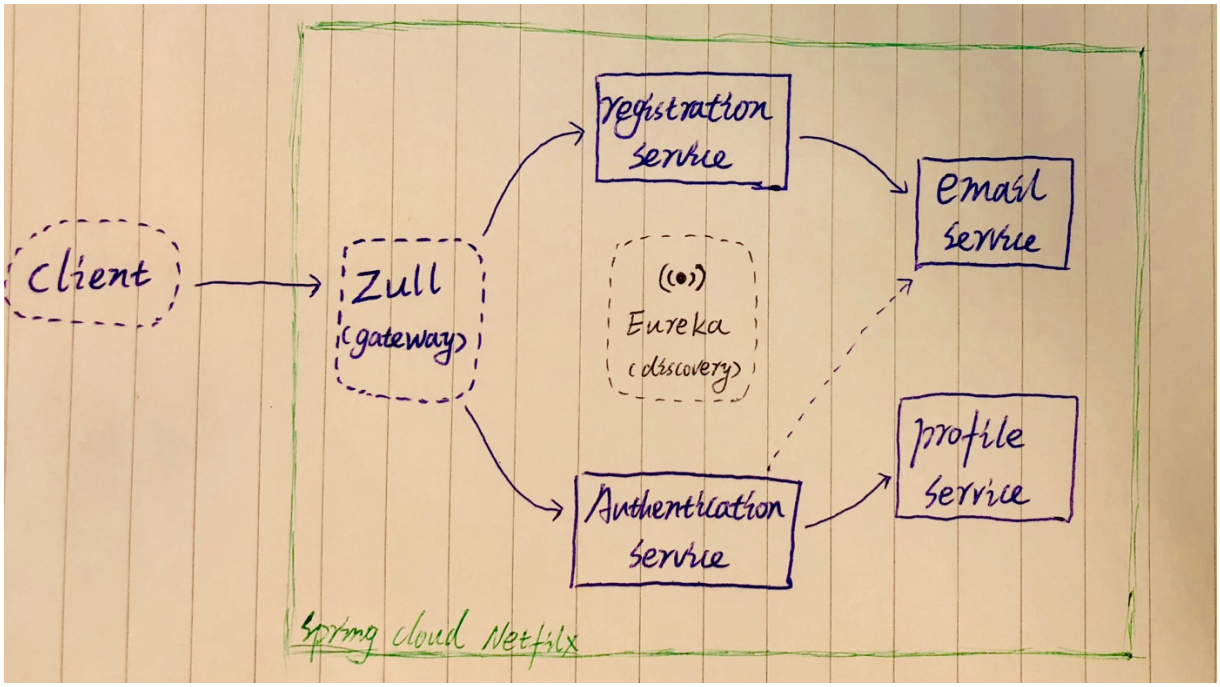


Figure 5 Server-side structure

This back-end server implements ***Spring Cloud Netflix*** framework. The *Eureka* discovery all services registered inside this structure. And the *Zuul* acts as the gateway, which means all client requests to access back-end services need through this gateway. In order to achieve the authentication and validation functions, we design that directing all requests from the client to Authentication server except requests for registration. Then accessing goal services (email service or profile service) when success authenticates with a valid role.

5.1.1 Eureka (service discovery)

Figure 14 Rureka Config UML

There is only one class in this Eureka discovery service, this class contains @SpringBootApplication, @EnableEurekaServer two annotations to declare this is a *spring boot* project and implements *EurekaServer* from *Spring Cloud Netflix* framework.

To enable the server, related dependencies is needed:

```
<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Dalston.SR2</version>
</parent>
<dependency>
        <!-- Setup Spring Boot -->
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
        <!-- Eureka service registration -->
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

Maven parent POM (or super POM) is used to structure the project to avoid redundancies or duplicate configurations using inheritance between pom files. It helps in easy maintenance in long-term. For example, since we use *spring-cloud-starter-parent*, we do not need to add an extra dependency like *spring-cloud-start*. *As mentioned earlier, different dependency version provides different support. We use version **Dalston.SR2** here, because we find that only this version can support Neo4j currently.(discuss details later) For the consistency of the *Micro Service Me Up* project, we use the same version of the spring cloud start parent.

Next step, configuring the server. In the *applications.yml* file:

```
# Configure this Discovery Server
server:
  port: 8085          # HTTP (Tomcat) port
eureka:
   server:            # nothing useful found for project yet
   instance:
     preferIpAddress: false
   leaseExpirationDurationInSeconds: 10
                      # indicates the time that eureka server waits since it received the last
                      # heartbeat before it can remove this instance from its view
     # eureka.instance.appname=${spring.application.name}
  client:             # Not a client, don't register with yourself
     registerWithEureka: false
     fetchRegistry: false
     region: default
     serviceUrl:
          defaultZone: http://ec2-54-201-208-226.us-west-2.compute.amazonaws.com:8085/eureka/
   # Notice that the serviceUrl is pointing to the same host as the local instance.
```

After every thing done, the main page generated by Eureka as follows:

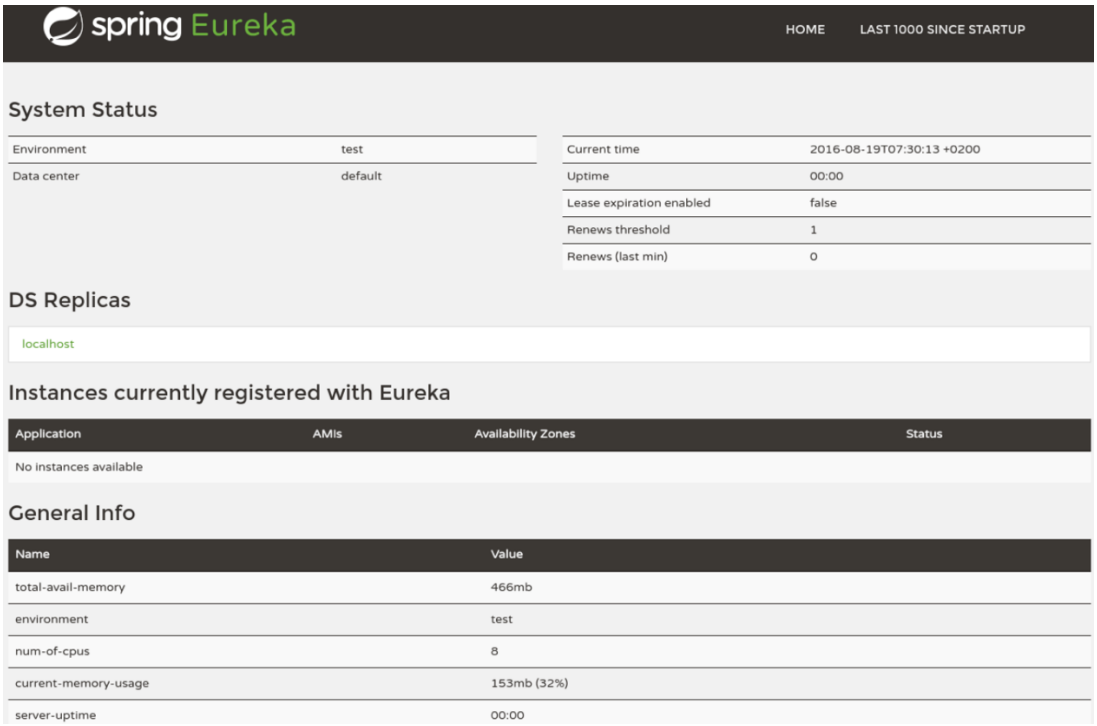


Figure 15 Eureka control page

Later, micro-services which connect to Eureka will be list as an instance on this page.

5.1.2 Zuul (gateway)

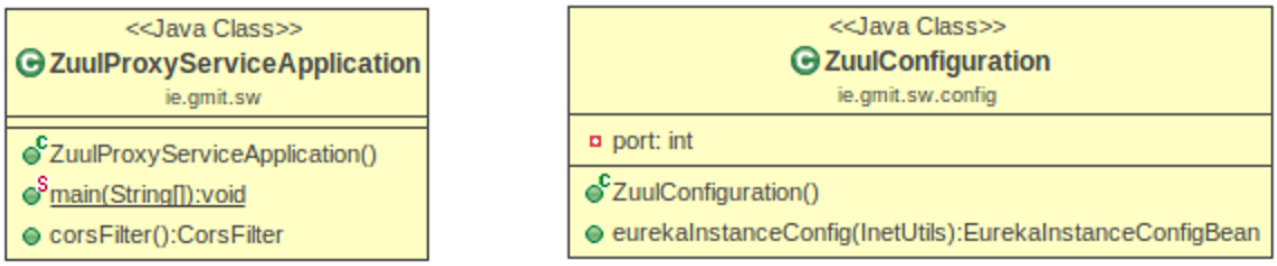


Figure 16 Zuul config UML

Similarly, the class – ZuulProxyServiceApplication contains @SpringBootApplication, @EnableDiscoveryClient, @EnableZuulProxy annotations to declare this *zuul server* is a spring boot application, a client of Eureka (that means it can be discovered by Eureka) and a gateway application. For the class – ZuulConfiguration, it is related public ips communicating between docker containers inside AWS.(more details illustrate in docker part)

In the POM file, all dependencies mentioned in Eureka is required (*spring-cloud-starter-parent* with version **Dalston.SR2**, and Eureka service registration). In addition, add spring-cloud-start-zuul to enable the gateway.

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

In the *applications.yml* file:

```
eureka:
  instance:
    prefer-ip-address: false # When the application registers with eureka, it will use its IP
Address rather than its hostname.
    leaseRenewalIntervalInSeconds: 5   # how often client send heartbeats to Eureka server
  client:
    #registerWithEureka: true      # true by default
```

```
    #fetchRegistry: true           # true by default
    serviceUrl:
      defaultZone: http://ec2-54-201-208-226.us-west-2.compute.amazonaws.com:8085/eureka/
                   # the ip address of eureka server set up on AWS.
hystrix:
  command:
    default:
      execution:
        isolation:
          strategy: THREAD
          thread:
            timeoutInMilliseconds: 60000
ribbon:
  ReadTimeout: 60000
  ConnectTimeout: 60000

zuul:
  #ignored-services: '*'
  routes:
    demo:                                    # inside the routes, next layer is the
      path: /demo/**                         # name of microservice.(demo, reg etc.)
      serviceId: DEMOMICROSERVER             #
    reg:                                     # path: → the goal router to access
      path: /reg/**                          # each server.
      serviceId: registration-service        # e.g. ZuulIP/reg/** → RegIP/**
    profile:                                 #
      path: /profile/**                      # serviceId is the name of the service,
      serviceId: profile-service             # can be found in the list of eureka
    email:                                   # server.
      path: /email/**
      serviceId: email-service
    auth:
      path: /auth/**
      serviceId: authentication-service
```

5.1.3 Registration Service

Functionality:

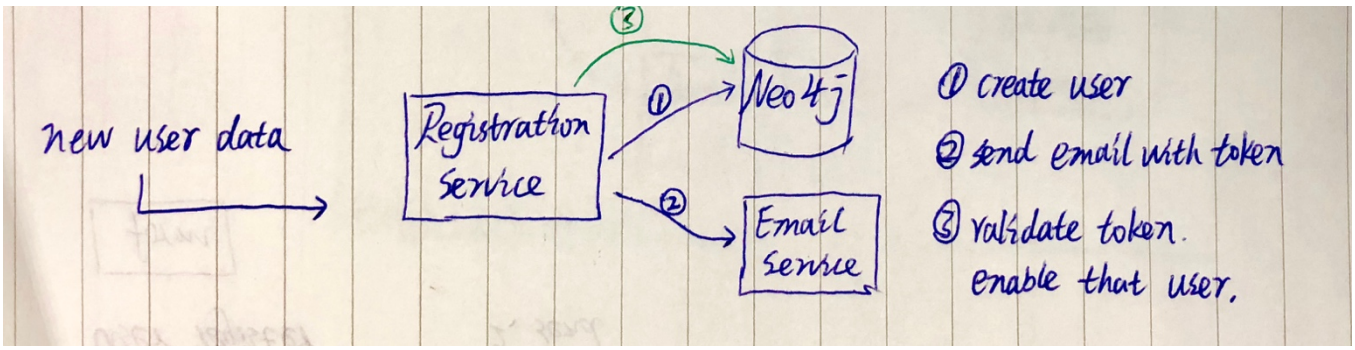


Figure 17 registration service structure

Once the form data received:

- Saving new user to the database (Neo4J) with the role (PendingRole).
- Generating a token and send it inside an email to user.
- When user click the validation link, change the user role to (NormalUser) in database and allowing login.

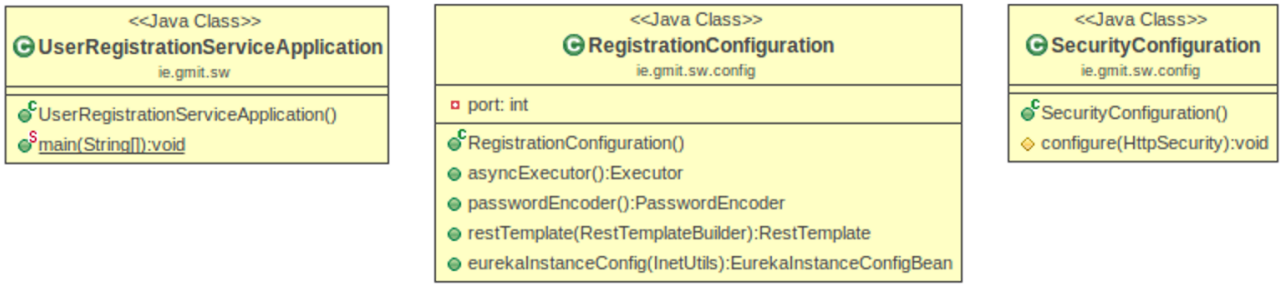


Figure 18 registration service config uml

UserRegistrationServiceApplication contains @SpringBootApplication, @EnableDiscoveryClient to declare it is a spring boot application and can be discovered by Eureka. *RegistrationConfiguration* relates to deploying docker on AWS. And SecurityConfiguration for permitting all routes. Why need this, because we implement *UserDetail* class which is a part of Spring Security to enable our new account validation.

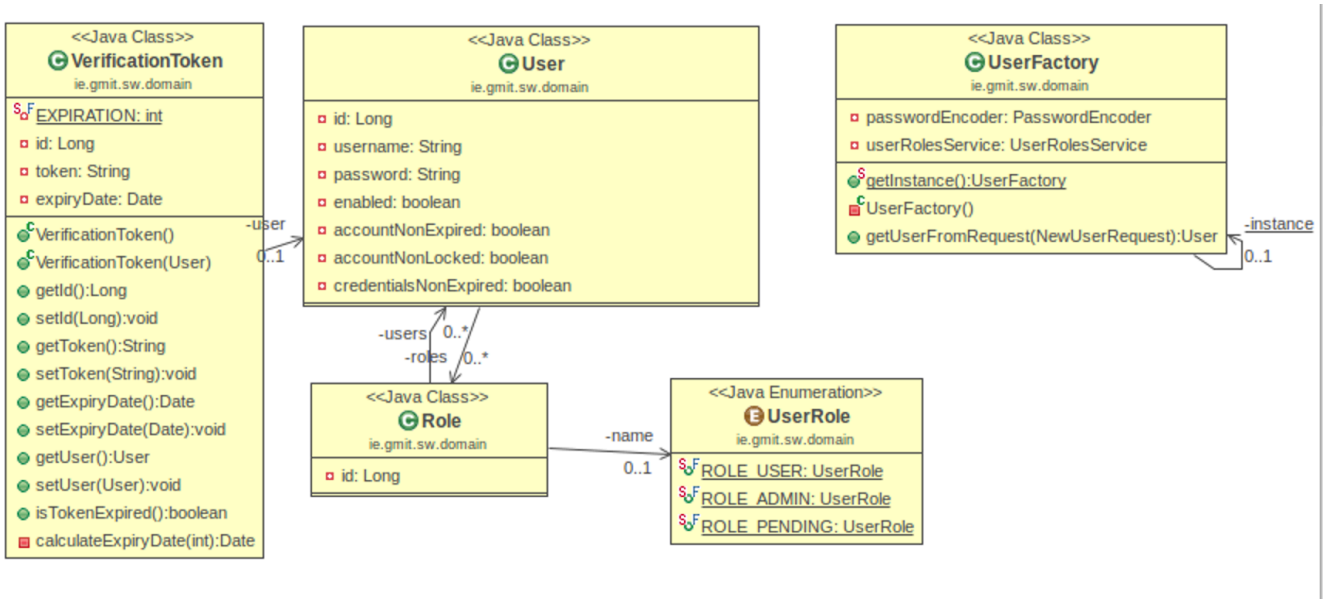


Figure 19 registration service model uml

Models: *VerificationToken* is used to check the user email is/not valid. It also contains expiryDate and we set it to 24 hours. A user can have multiple roles such as Admin and Normal user. Need to note that pending user are not allow to login. *UserFactory* just follow the Factory design pattern and Singleton design pattern and using `PasswordEncoder` of spring security to encrypt the password when create the user instance.

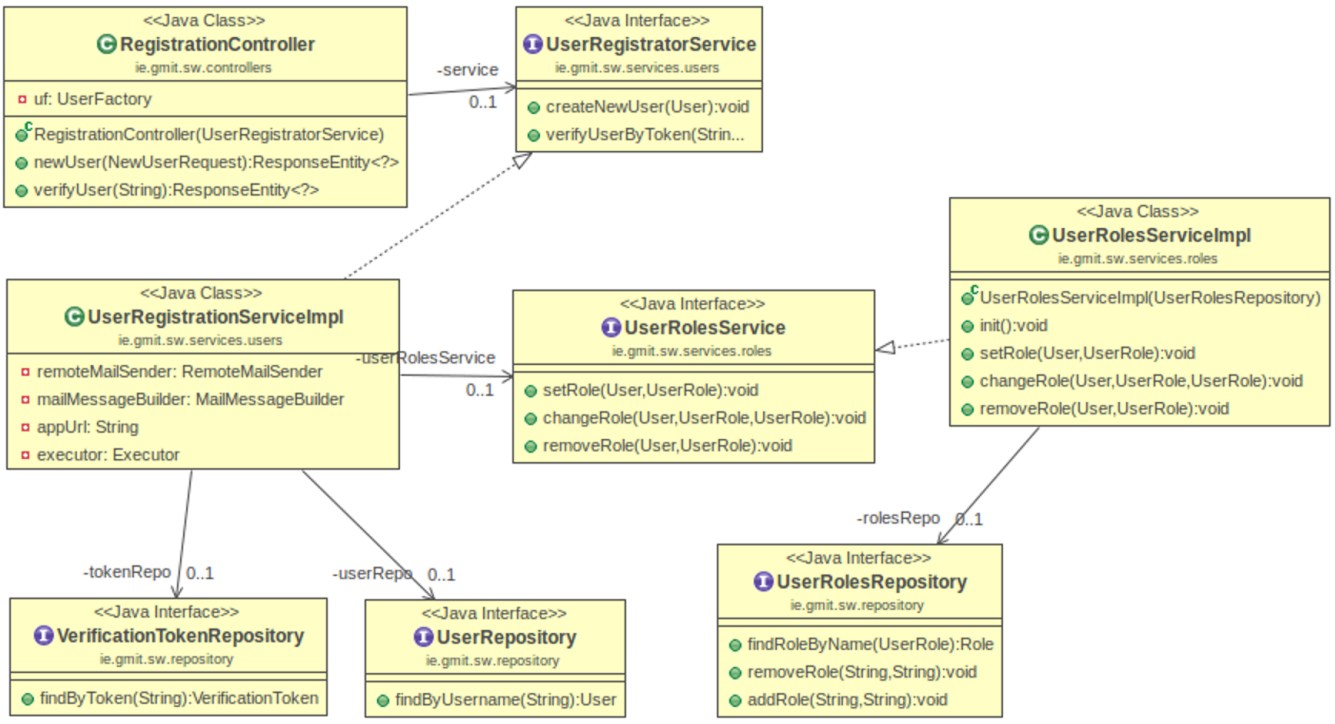


Figure 20 registration service controller uml

Basic spring model: Controller → Service → Repository. *UserRegistratiorService* create a new user and generates the token when a new user info received. It also provides function to verify the token stored in the database. In order to make sure the password is secure, we use When validation process is finished, change such user role to normal user and delete the token in database.

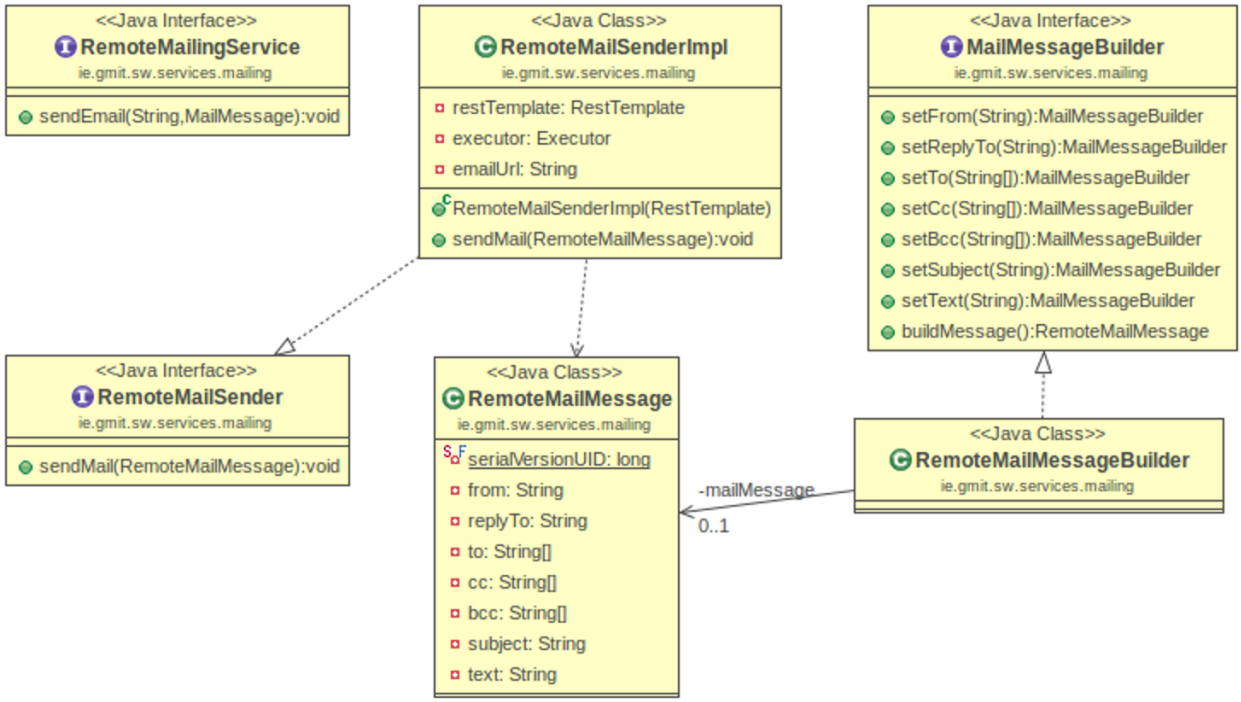


Figure 21 registration service mail message prototype uml

This part designs to generate a mail message object as a parameter to call Email service methods, it is included in the validation process. This is a full hierarchy structure for generating a remote mail message. These should be packaged in a jar and reused again and again when the Email Server is requested.

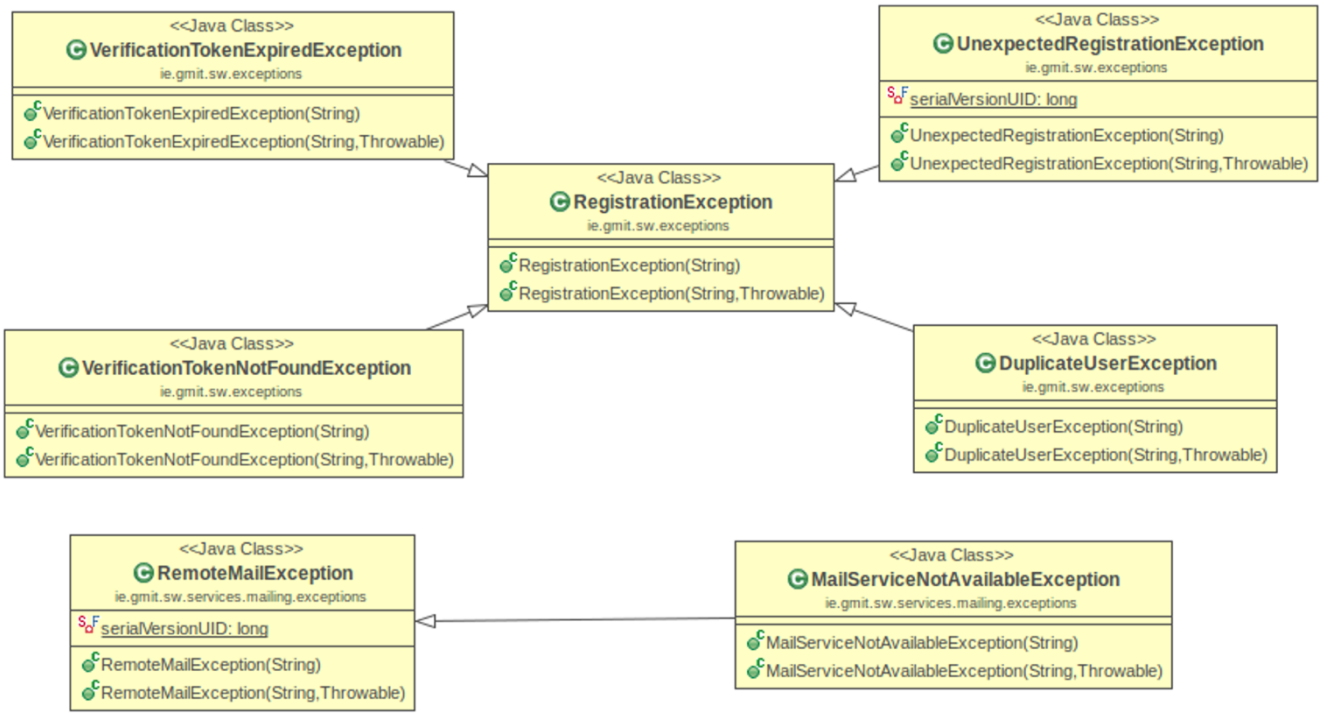


Figure 22 registration service exception uml

Custom exceptions, it provides information or functionality that is not part of Java's standard exceptions.

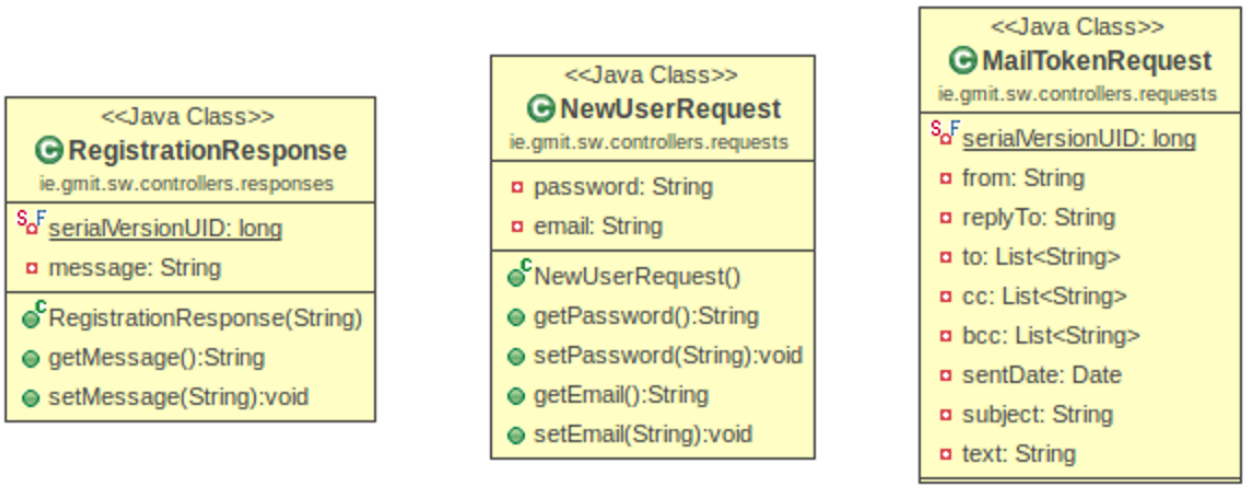


Figure 23 registration service req-resp uml

The prototype of requests and response to registration service. The common data type is good for clearly API invocation. Also, relevant exception message will send back to the user as a response object. For example, we enabled HttpStatus.ACCEPTED/CREATED/CONFLICT/BAD_REQUEST etc. as the response object.

In the POM file, except the dependencies Eureka needed, need to add:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-neo4j</artifactId>          # Support Neo4J database
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>                     # Support password encrypting
        <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

In the *applications.yml* file:

```
eureka:
  instance:
    prefer-ip-address: false # When the application registers with eureka,
                             # it will use its IP Address rather than its hostname.
    leaseRenewalIntervalInSeconds: 5   # How often client send heartbeats to Eureka server
  client:
    serviceUrl:
      # It is possible to configure service urls so that client uses closest zone of availability
      # Check for references: https://github.com/spring-cloud/spring-cloud-netflix/issues/1327
      # defaultZone: http://localhost:8085/eureka/ # address of an Eureka Server
      defaultZone: http://ec2-54-201-208-226.us-west-2.compute.amazonaws.com:8085/eureka/

gateway:
 gateway-path: http://ec2-54-201-208-226.us-west-2.compute.amazonaws.com:8086
 email-path: http://ec2-54-201-208-226.us-west-2.compute.amazonaws.com:8086/email/sendemail/
# gateway is for redirecting in zuul server.
```

*NOTE: database configuration is describes in darabase part.

APIs:

- **Register**

Request

Method	URL
POST	api/user/new

Type	Params	Values
HEAD	application/json	string
POST	email	string
POST	password	string

Response

Status	Response

201	{
	"message": "register success!"
	}

409	{" message ":"the user email isalready exist!"}
400	{" message ":"Unexpected error."}

Figure 24 register api

- **verify token**

Request

Method	URL
GET	api/verify/{token}

Type	Params	Values
HEAD	application/json	string
GET	token	string

Response

Status	Response

202	{
	"message": "Account verified. You may login."
	}

400	{" message ":"Unexpected error."}

Figure 25 token verify api

5.1.4 Email Service

The main functionality for this server is sending the email. Currently, we use this service to send the email with verify token to new user. [8]

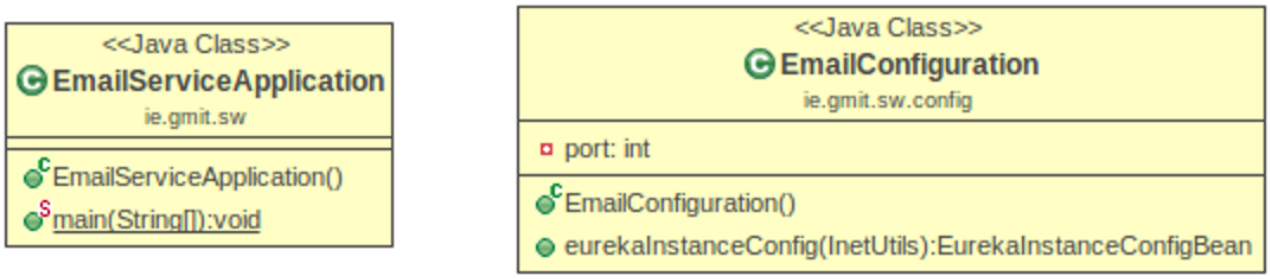


Figure 26 email service config uml

Similarly, *EmailServiceApplication* includes @SpringBootApplication, @EnableDiscoveryClient to declare it is a spring boot application and can be discovered by Eureka. *EmailConfiguration* handles communication with deploied docker on AWS.

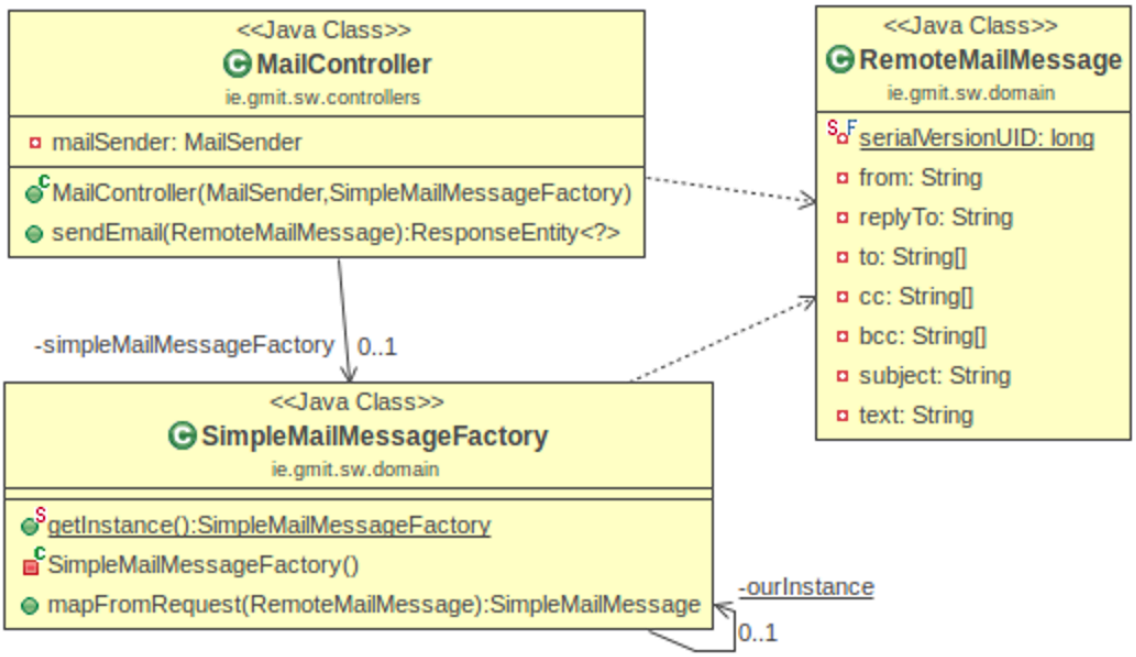


Figure 27 email service controller uml

Simply send email. Remember the RemoteMailMssage object is generated already and as the parameter when calling this Email server (the full hierarchy structure described in registration part).

In POM file, nothing special but email dependency is needed:

```
<dependency>
        <!—add email dependency -->
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

In the *applications.yml* file:

```
eureka:
  instance:
    prefer-ip-address: false              # When the application registers with eureka.
                                          # it will use its IP Address rather than its hostname.
    leaseRenewalIntervalInSeconds: 5      # How often client send heartbeats to Eureka server
  client:
    serviceUrl:
      # It is possible to configure service urls so that client uses closest zone of availability
      # defaultZone: http://localhost:8085/eureka/ # for local testing
      defaultZone: http://ec2-54-201-208-226.us-west-2.compute.amazonaws.com:8085/eureka/
```

APIs:

- **sending email**

Request

Method	URL
POST	api/sendemail

Type	Params	Values
HEAD POST	application/json remoteemail	string RemoteMailMessage

RenoteMailMessage: the normal object contains info required when sending a email, such as email title/content, sent to etc. (the full hierarchy structure described in registration part).

Response

Status	Response

200	{
	"message": "Email send successfully!"
	}

400	{" message ":"Unexpected error."}

Figure 28 email sending api

5.1.5 Profile Service

This *profile service* is used to store user details like user nickname, gender etc. After discussing with superviser that this might be too simple to demonstrate to the user, we decide to add extra service items. Generally, *Items service* should be another microservice. Because the limitation of AWS instance we have, we are unable to run more projects, we conbine the service item function and the profile service togrther.

Functionality:

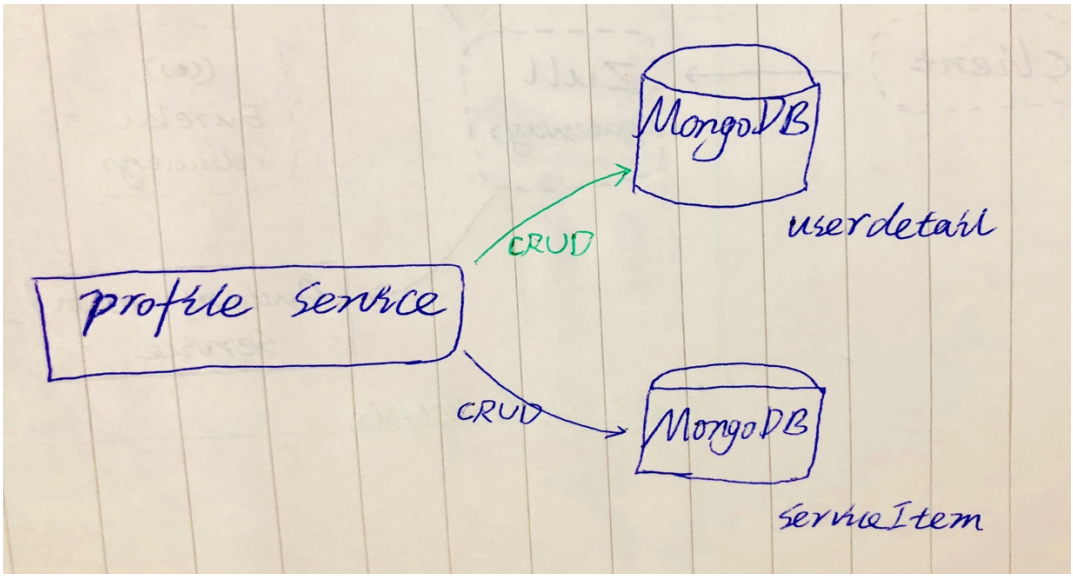


Figure 29 profile service structure

This server using MongoDB handles CURD (create/update/retrieve/delete) to user and item objects in one database server but different databases.

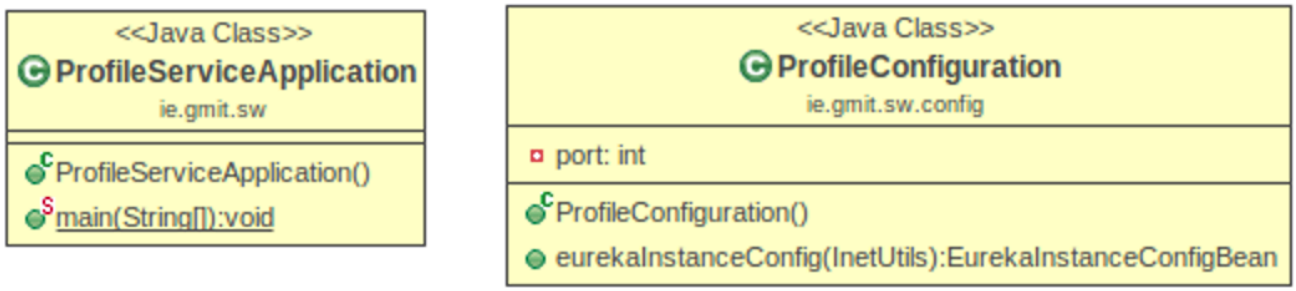


Figure 30 profile service config uml

Similarly, *ProfieServiceApplication* includes @SpringBootApplication, @EnableDiscoveryClient to declare it is a spring boot application and can be discovered by Eureka. *EmailConfiguration* handles communication with deploied docker on AWS.

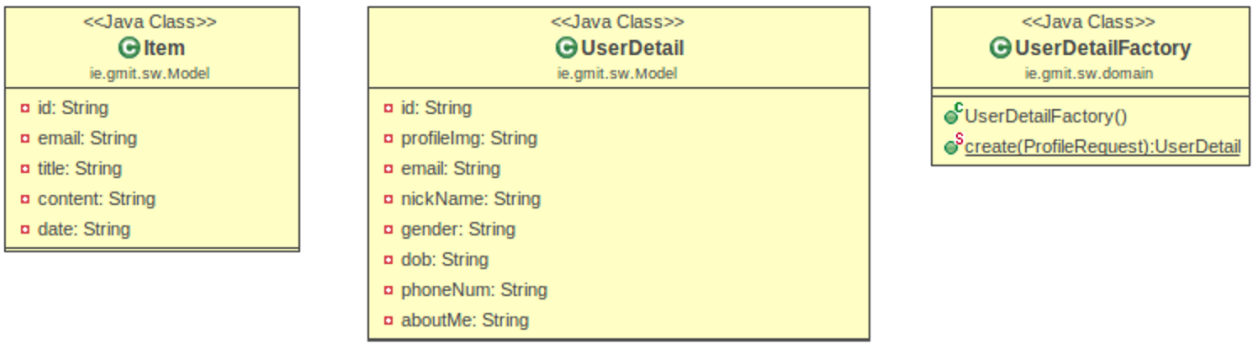


Figure 31 profile service model uml

UserDetailFactory is using factory and singleton design pattern to create objects more secure and efficient. As designed earlier, item is add-on part, so no itemFactory.

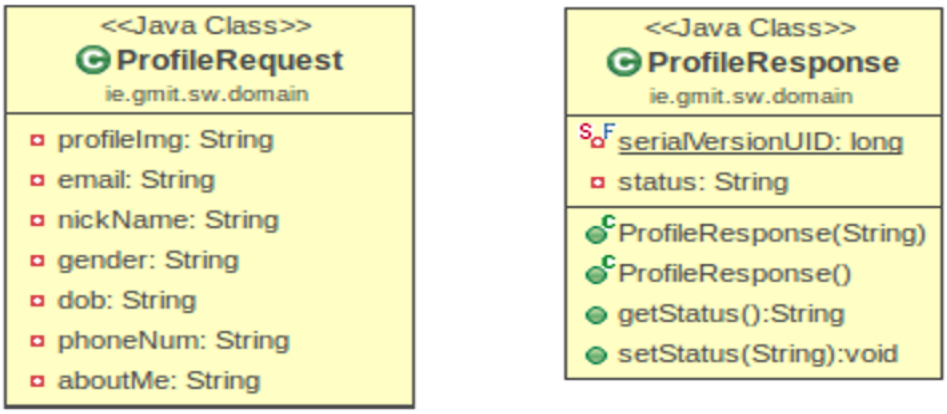


Figure 32 profile service req-resp uml

The prototype of requests and response to profile service for keeping consistency.

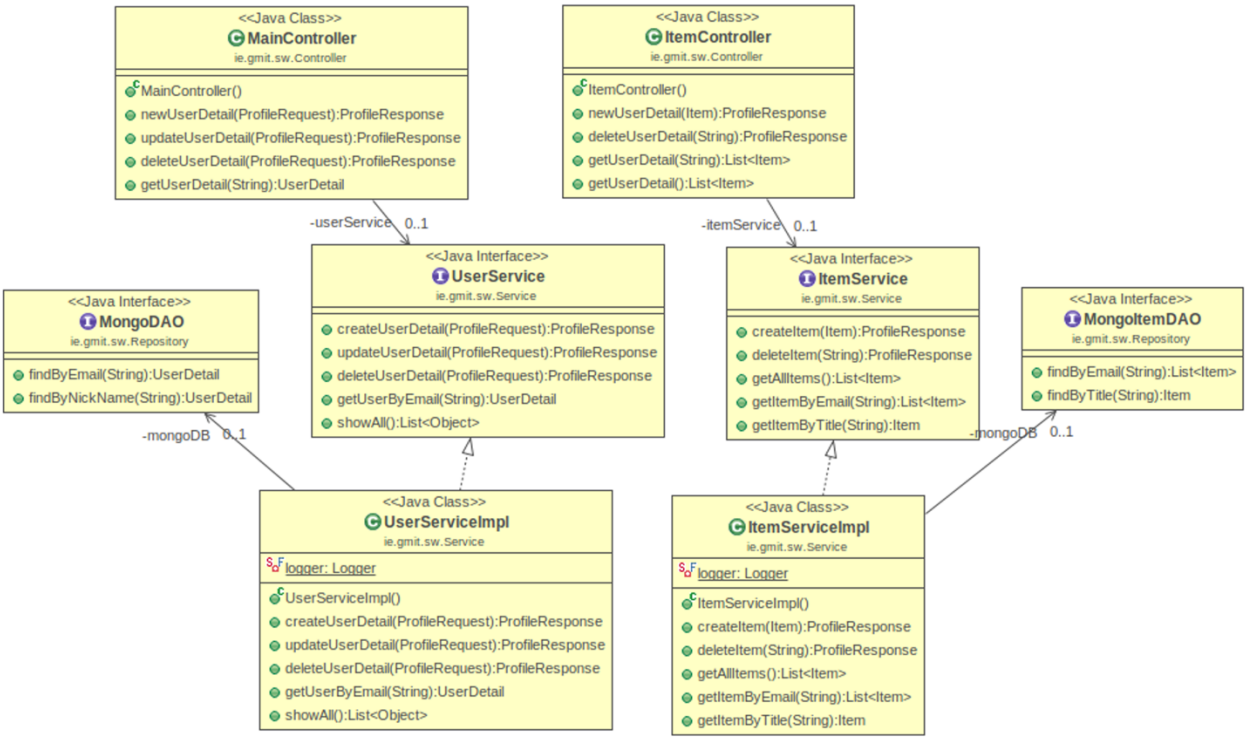


Figure 33 profile service mian uml

Basic spring model: Controller → Service → Repository. MainController handles user details object, ItemController handles item object.

In the POM file, except the dependencies Eureka needed, need to add:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>          # Support MongoDB database
</dependency>
```

In the *applications.yml* file:

```
eureka:
  instance:
    prefer-ip-address: false              # When the application registers with eureka,
                                          # it will use its IP Address rather than its hostname.
    leaseRenewalIntervalInSeconds: 5      # How often client send heartbeats to Eureka server
  client:
    serviceUrl:
      # defaultZone: http://localhost:8080/eureka/ # local testing
      defaultZone: http://ec2-54-201-208-226.us-west-2.compute.amazonaws.com:8085/eureka/
```

*NOTE: database configuration is describes in darabase part.

APIs:

- **UserDetail CURD**

Request

Method	URL
POST	api/userdetails/new
POST	api/userdetails/update
POST	api/userdetails/delete
GET	api/userdetails/get

Type	Params	Values
HEAD	application/json	string

POST	ProfileRequest	ProfileRequest

ProfileRequest: the prototype object contains info both UserDetail and Item object attributes.

Response

Status	Response

200	{
	"message": "create success!" # for create user
	"update success!" # for update user
	"delete success!" # for delete user
	"userdetail": user object # for get user
	}

409	{" message ":"userdetail not exist / duplicated ."}
400	{" message ":"Unexpected error."}

Figure 34 user detail CURD api

- **Item CURD**

Request

Method	URL
POST	api/item/new
POST	api/item/update
POST	api/item/delete
GET	api/item/get

Type	Params	Values
HEAD	application/json	string
POST	ProfileRequest	ProfileRequest

ProfileRequest: the prototype object contains info both UserDetail and Item object attributes.

Response

Status	Response

200	{
	"message": "create success!" # for create item
	"update success!" # for update item
	"delete success!" # for delete item
	"userdetail": item object # for get item
	}

409	{" message ":"item not exist."}
400	{" message ":"Unexpected error."}

Figure 35 item CURD api

5.1.6 Authentication Service

This *Authentication Service* [9] handles the user account verification by using ***OAth2*** protocol for authorization. All requests for our services which need a valid user role goes to this *Authentication Service* first. After the verification process, the user info will be stored in the ***XSRF-TOKEN*** token and back to request sender.

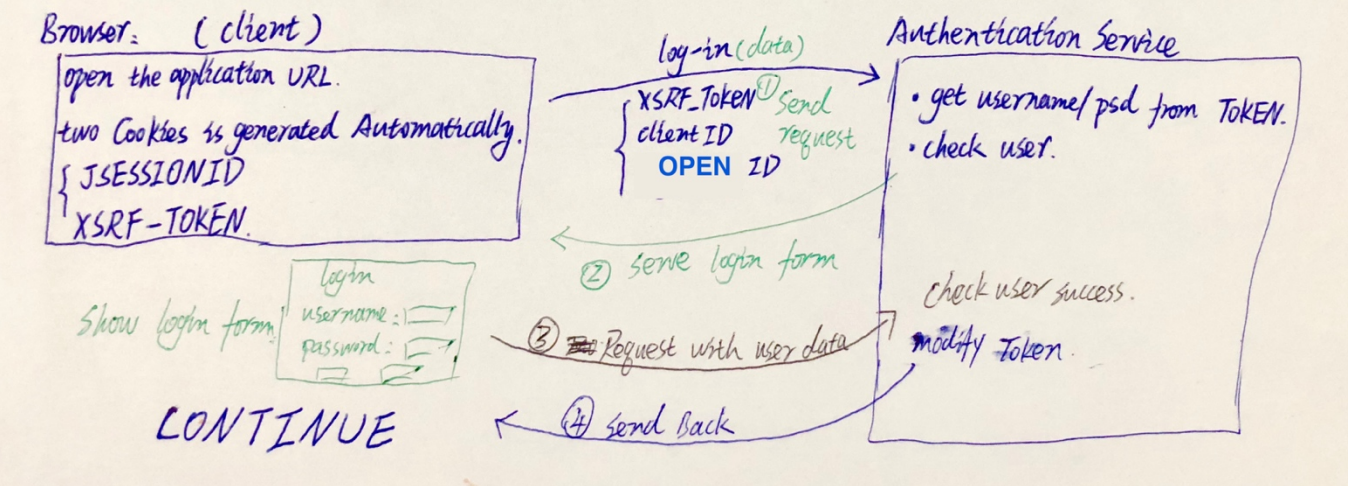


Figure 36 authentication service structure

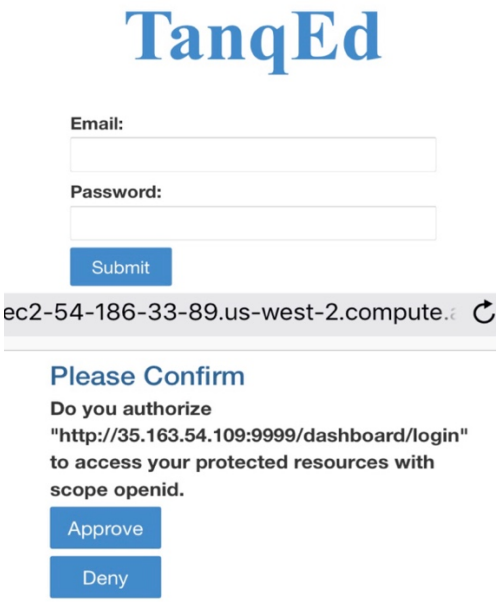


Figure 37 login page show

This *Authentication Service* provide login page. [layout using *Bootstrap* provide by *Webjar* in macen spring boot project] Which means all client service can use this login page directly. Also it is possiable to add this into a pop window for login in the client applications.

UMLs:

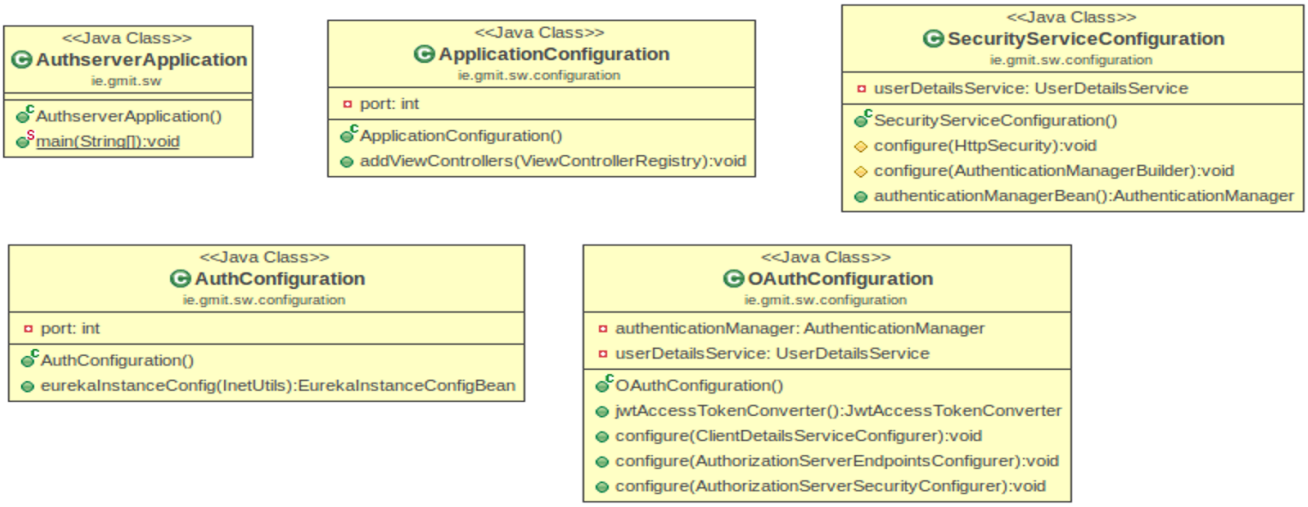


Figure 38 authentication service config uml

Similarly, *AuthenticationApplication* includes @SpringBootApplication, @EnableDiscoveryClient to declare it is a spring boot application and can be discovered by Eureka. *AuthConfiguration* handles communication with deploied docker on AWS. ApplicationConfiguration, SecurityServiceConfiguration and OAuthConfiguration handles converting token, configuring security things etc.

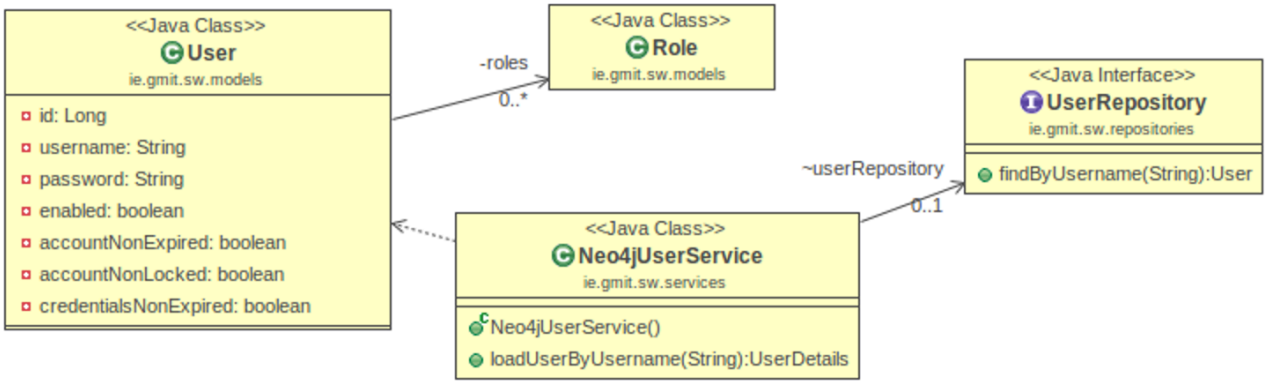


Figure 39 authentication service model uml

This role is the same role inside the *Registration Service*. That means the Authentication Services can author specific services for different user roles.

In POM file, these dependencies are ewquired:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.security.oauth</groupId>
        <artifactId>spring-security-oauth2</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-jwt</artifactId>
</dependency>

<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-neo4j</artifactId>
</dependency>
# this security related dependencies is the key of the service
# put neo4j here, because this server need to check user with the user stored in Neo4j database
```

In the *applications.yml* file:

```
eureka:
  instance:
    prefer-ip-address: false              # When the application registers with eureka,
                                          # it will use its IP Address rather than its hostname.
    leaseRenewalIntervalInSeconds: 5      # How often client send heartbeats to Eureka server
  client:
    serviceUrl:
      # defaultZone: http://localhost:8080/eureka/ # local testing
      defaultZone: http://ec2-54-201-208-226.us-west-2.compute.amazonaws.com:8085/eureka/
```

*NOTE: database configuration is describes in darabase part.

APIs:

- **Verify**

Request

Method	URL
GET	api/uaa/login

Type	Params	Values
HEAD	***XSRF-TOKEN*** token	string

ProfileRequest: the prototype object contains info both UserDetail and Item object attributes.

Response

Status	Response
200	1) LoginPage.html 2) data with updated token
400	{"Unexpected error."}

Figure 40 authentication service validation api

5.2 Front-end

5.2.1 Register Page Handler

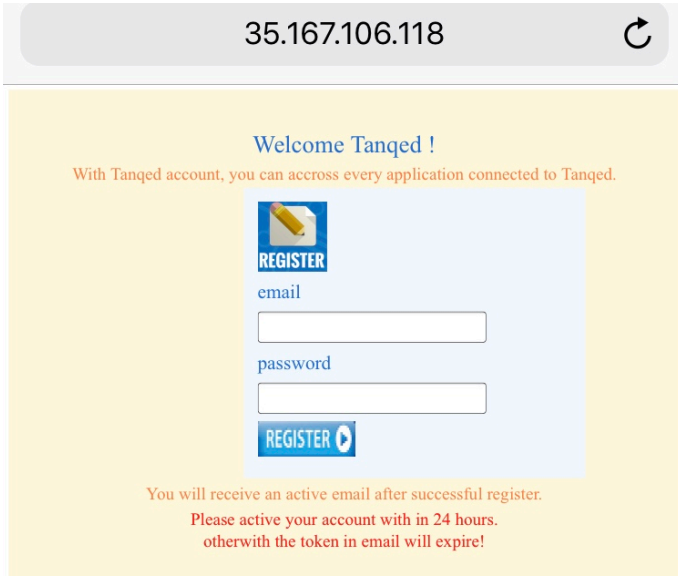


Figure 41 register page show

This part is a SPA (Simple Page Application) just to handle registration. In order to simulate openID principle, we design this page to show this part can be separately implementing by outher platform who wants to use *Tanqed Account*.

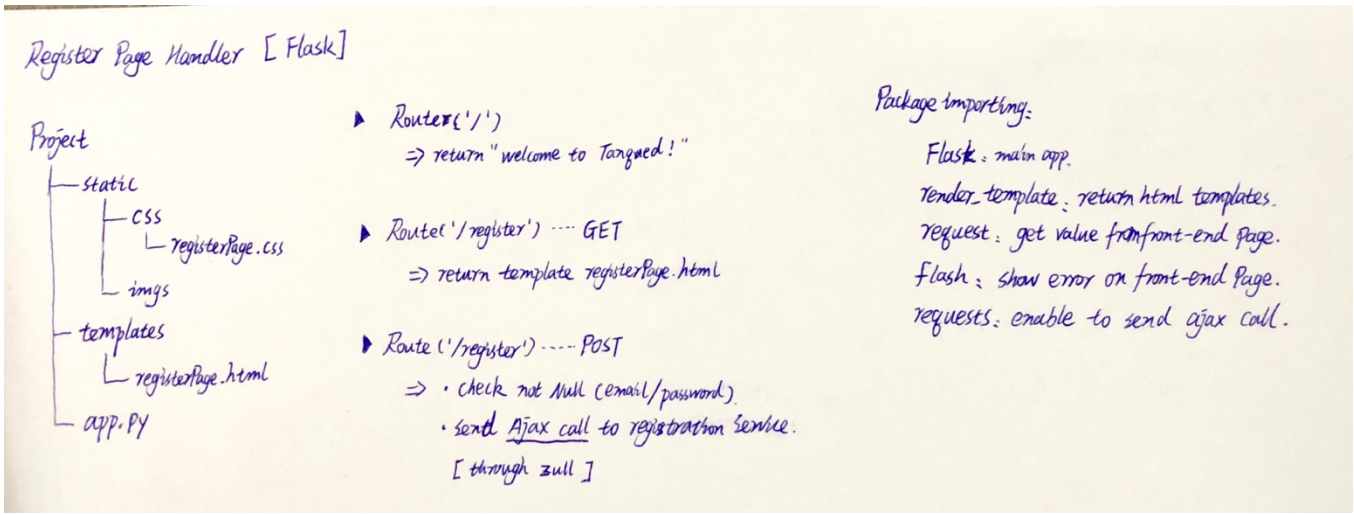


Figure 42 flask spa structure

For this log-in & register part, some packages are be used:

- Render_Template

  To render a template, all you have to do is provide the name of the template and the variables you want to pass to the template engine as keyword arguments. Every html file will be returned correctly by this method.
- Request.Form

The Form collection retrieves the values of form elements posted to the HTTP request body, with a form using the **post** method. This method will transmit the data you typed like username, password to the back-end.

- Flush

  Flask provides a really simple way to give feedback to a user with the flashing system. The flashing system basically makes it possible to record a message at the end of a request and access it next request and only next request. Here it is used to display the error messages like "wrong username/password", "user existed" etc.

- Redirect

  It returns a response object and redirects the user to another target location with specified status code. When you logined successful it will redirect you to public chat page.

- Requests

  Handle different HTTP methods. For example, using *requests.get* method to send username/password to the server for log-in, and *requests.post* method for register communications.

5.2.2 Mobile Client

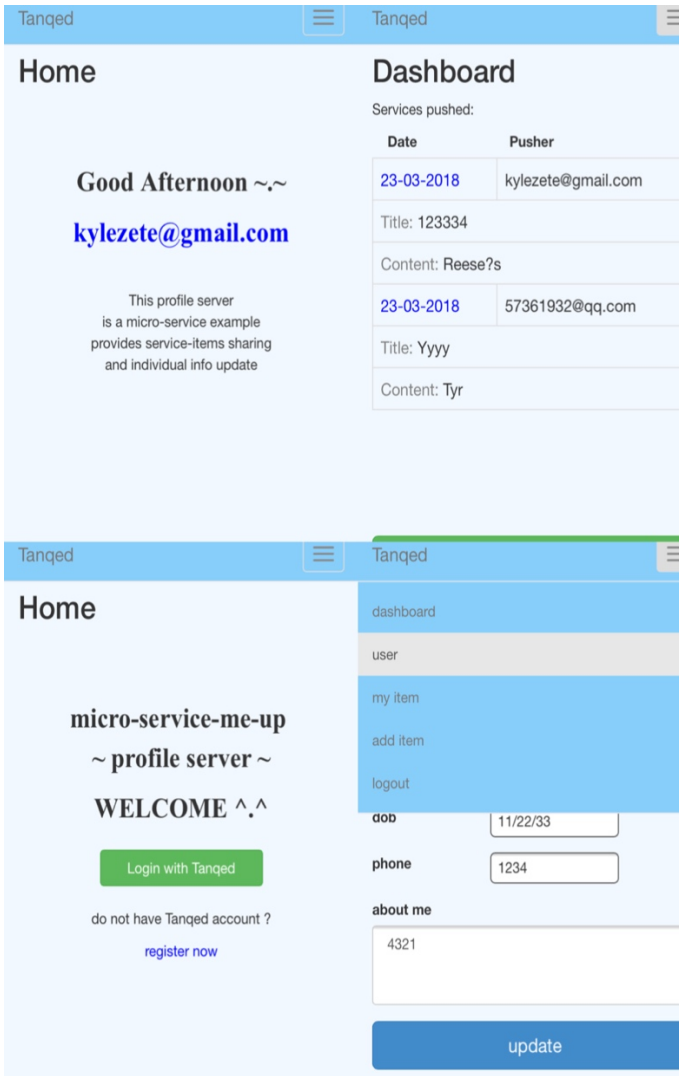


Figure 43 mobile client demo show

This application is design for users who using mobile devices. Considering multiple device system platforms such as ios, android, windows phone etc. we decided to build a web application to avoid the compiling process for each platform.

This application enables user to modify user details and items.

- Project framework: **Spring Boot**
- Page design: **AngularJS** & **Bootstrap**
- Security: **Spring Security** & **OAth2** with **AngularJS**

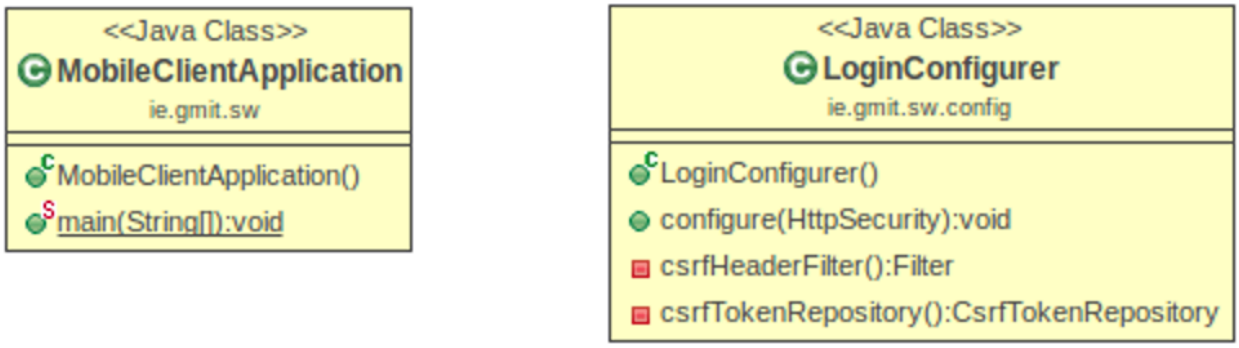


Figure 44 monile client config uml

MobileClientApplication contains @SpringBootApplication to declare it is a spring boot application. This application is not a part of Spring Cloud Server, that is why do not have relevant configuration file for setting docker public IP.

LoginConfiguration extends WebSecurityConfigurerAdapter, handling security stuff (OAth2), achiving only valid logined user can access other services. ***XSRF-TOKEN*** is created and store to *cookies*.

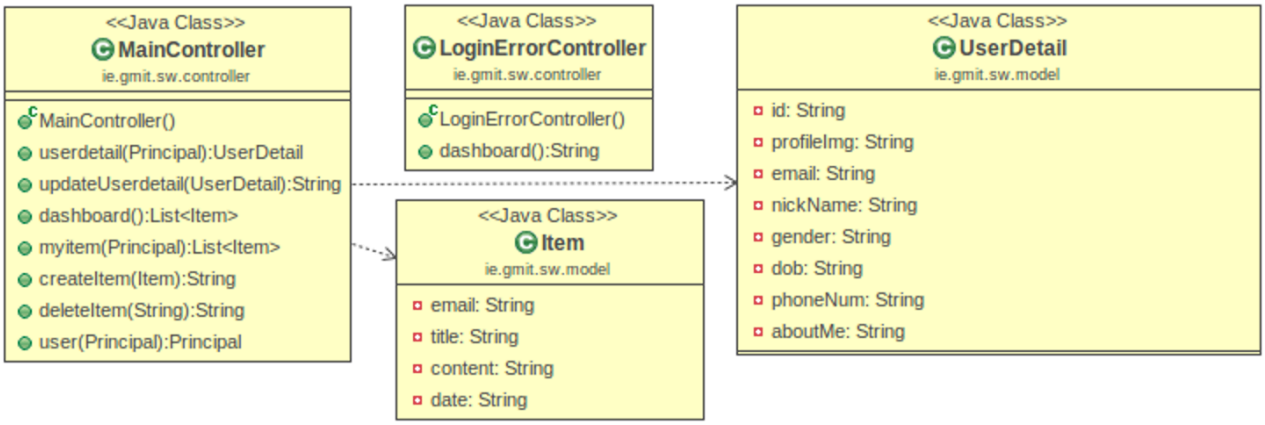


Figure 45 mobile client main uml

The main function of these components are receiving ***Ajax*** call from from front-end page (sending by ***AngularJS***), and generate a remote call to other microservices to get services.

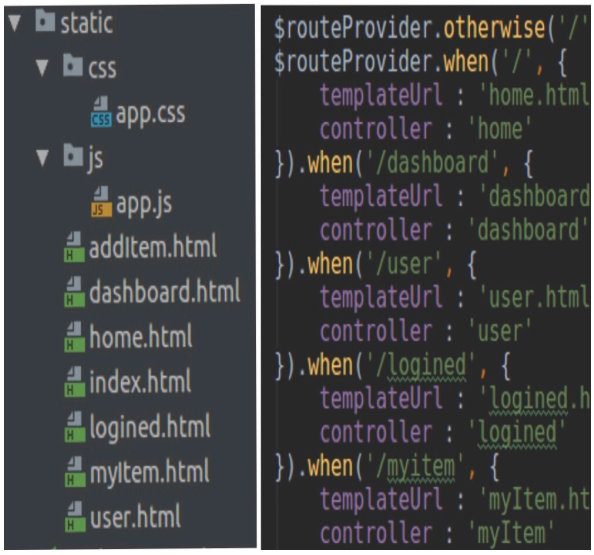


Figure 46 mobile client angular sample

AngularJS controls the page accessability. Once the user role is checked, set another options in the manu to visiable.

Angular ng-model directive binds the value of HTML controls (input, select, textarea) to application data. ***Angular scope*** is an object with the available properties and methods. ***Angular ng-controller*** is defined by a JavaScript **constructor function** that is used to augment the ***Angular Scope.***

In POM file, we need add dependencies: *spring-boot* for declare the spring boot application, *oath2 & spring security* to support authentication and validation function, *angularjs* to enable anfular script, *bootstrap* to enable bootstrap layout, *httpclient* to communicate with angular and send request to extra services.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

```
<dependency>
        <groupId>org.webjars</groupId>
        <artifactId>angularjs</artifactId>
</dependency>
<dependency>
        <groupId>org.webjars</groupId>
        <artifactId>bootstrap</artifactId>
</dependency>
<dependency>
        <groupId>org.apache.httpcomponents </groupId>
        <artifactId> httpclient </artifactId>
</dependency>
```

*NOTE: in spring, angularJS and bootstrap are provided by ***org.webjars***

In the *applications.yml* file:

```
security:
  user:
    password: user
  ignored: /,/favicon.ico,/index.html,/home.html,/dashboard.html,/js/**,/css/**,/webjars/**
  sessions: ALWAYS
```

```
  oauth2:
    sso:
      loginPath: /dashboard/login                    # Basic spring security configurations
management:
  security:
    role: HERO
logging:
  level:
    org.springframework.security: DEBUG
    com.netflix.discovery: 'OFF'
---------------------------------------------------------------------------------------------------------------
spring:
  profiles: default
security:
  oauth2:
    client:
      accessTokenUri: http://ec2-54-186-33-89.us-west-2.compute.amazonaws.com:8090/uaa/oauth/token
      userAuthorizationUri: http://ec2-54-186-33-89.us-west-
2.compute.amazonaws.com:8090/uaa/oauth/authorize
      clientId: acme                                 # connect to Authentication server
      clientSecret: acmesecret                       # requires clientId → client application id
    resource:                                        # jwt key: here we just use open public key
      jwt:
        keyValue: |
          -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnGp/Q5lh0P8nPL21oMMrt2RrkT9A
W5jgYwLfSUnJVc9G6uR3cXRRDCjHqWU5WYwivcF180A6CWp/ireQFFBNowgc5XaA0kPpzEtgsA
5YsNX7iSnUibB004iBTfU9hZ2Rbsc8cWqynT0RyN4TP1RYVSeVKvMQk4GT1r7JCEC+TNu1ELmb
NwMQyzKjsfBXyIOCFU/E94ktvsTZUHF4Oq44DBylCDsS1k7/sfZC2G5EU7Oz0mhG8+Uz6MSEQH
toIi6mc8u64Rwi3Z3tscuWG2ShtsUFuNSAFNkY7LkLn+/hxLCu2bNISMaESa8dG22CIMuIeRLVcAm
EWEWH5EEforTg+QIDAQAB
          -----END PUBLIC KEY-----
      id: openid
      serviceId: ${PREFIX:}resource
```

5.2.3 PC Client

To show variety of client technologies available to work with our microservices system, we decided to build application using ReactJS front-end library for PC users to access services within their browser.

Platform:

- NodeJS – base platform to load, compile and run JavaScript applications.

Libraries:

- ReactJS
    1. react – ReactJS base library
    2. react-dom – ReactJS library that builds and manipulates virtual DOM
    3. react-router – ReactJS library for dynamic application routing
    4. reactstrap – bootstrapped components library for layout and design of an application. Using latest Bootstrap 4.
- Bootstrap 4 – required for reactstrap library
- jQuery – required for Bootstrap library for dynamic classes like menu toggling
- popper.js – required by Bootstrap 4 library for dynamic classes like popping components#

ReactJS Basics

React classes – representing React components. Depending on whether a state is needed inside component, we can code component in two ways:

Stateful component

1. To be a React component, your class has to extend Component imported from React library. This allows you to override render() function to return html template in form of JSX.
2. Construction method constructor() defines the state of the component and also binds other functions written inside component.
3. render() method where we place html block as JSX. We can access component state and other defined functions inside JSX using {this.___} syntaxis.
4. Important to note that only way to modify state values from JSX is by using handling function bound to an input field and use setState method from Component class.

Usage Example:

```
// import main React component
import React, { Component } from 'react';

class SampleClass extends Component{
    constructor(props){
            super(props);
            this.state = {
                        age = 0,
                  name = 'default name'
            }
            this.handleInput = this. handleInput.bind(this);
    }

    handleInput(event){
            const target = event.target;
            const name = target.name;

            this.setState({
                [name]: value
            });
    }
    render(){
            return(
             <input onChange = { this.handleInput }/>
                <h1>Hello, {this.state.var2}<h1>
            );
    }
}
```

Stateless Component

1. To make a stateless functional React component it is enough to import React into module.
2. Defined constant in form of a function can accept parameters a treat them as normal props passed inside component.
3. Normal variables and functions can be passed as a props to a functional component.

Usage Example:

```
// import main React component
import React, { Component } from 'react';

const FunctionalComponent = ({name, age, handleInput}) => (
            <input onChange = { handleInput }/>
            <input onChange = { handleInput }/>
            <p>Name: {name}, Age: {age}<p>
);

export default FunctionalComponent;
```

About Props

They only way to pass state from one component into another component is by means of props.

At the construction time of the component it will accept props where they can be used to either initialize state or render values.

Passing props into component related to previous example:

```
….
<MyComponent handleChange = {this.handleChange}
                        name = {this.state.name}
                    age = {this.state.age}/>
….
```

Main Application Components

index.js – a root component of React application. Renders a parent component of components hierarchy. Used for initial set up of an application like importing other libraries, setting up a store and other process related to an application start-up.

Application-router – A parent component for an application. Renders a base structure of an app, including header container, body content container and footer container if desired. Performs a dynamic routing to another components within application with *react-router* library.

Three important components to know from *react-router*:

- *BrowserRouter* – this is the Router itself. Uses HTML5 history API. Synchronizes UI content to what is written in address bar (URL).
- *Route* – links components to its respective URI. Responsible for rendering of the UI.
- *Link* – wrapper for html link tag *<a>* that is used for redirect.

Additional libraries to use with *react-router:*

createBrowserHistory – Comes with *react-router* as a stand alone library. This library making sure that application history is maintained consistently in respect to a user actions. Added as a parameter to a router component *BrowserRouter*.

Usage Example:

```
import React from "react"
import {BrowserRouter, Route, Link} from "react-router-dom"
import createBrowserHistory from "history/createBrowserHistory"
import Home from './home';

// Create new browser history to use in Router
const history = createBrowserHistory();

// Other component to navigate to
const Home = () => (
   <Home/>
)

class Navigation extends React.Component {
   render() {
      return (
            // Router itself
         <BrowserRouter history ={history}>
            <div>
               <ul>
                  <li>
                     <Link to="/">Home</Link>
                  </li>
               </ul>
            </div>
            // Links url pattern to specific component, Home in this case that is defined above.
            <Route exact path="/" component = {Home}/>
         </BrowserRouter>
      )
   }
}
export default Navigation
```

Navigation-bar.js – component to render navigation bar and laout a Links to navigate across an application. Uses *reactstrap* library with bootstrapped components for navigation bar styling. Performs component switching logic based on authentication status of the user, i.e., if not authenticated, render login or signup links, otherwise profile and signout links.

Usage Example:

```
import React from "react"
import { NavLink as Link} from "react-router-dom"
import { Navbar,  NavbarBrand,
          Nav,  NavItem,  NavLink } from 'reactstrap';

// Functional component that can be rendered later within return scope of React render() function

const GuestPanel = () => (

// Section of the navigation bar containing navigation items
   <Nav className="ml-auto" navbar>
      <NavItem>
         <NavLink to="/registration" activeClassName="active" tag={Link}>Sign Up</NavLink>
      </NavItem>
      <NavItem>
         <NavLink to="/login" activeClassName="active" tag={Link}>Login</NavLink>
      </NavItem>
   </Nav>

);

const ServicePanel = () => (
   <Nav className="mr-auto" navbar>
      <NavItem>
         <NavLink to="/services" activeClassName="active" tag={Link}>Services</NavLink>
      </NavItem>
      <NavItem>
         <NavLink to = "/contacts" activeClassName="active" tag={Link}>Contacts</NavLink>
      </NavItem>
   </Nav>
);

// Composite function that return final product of the component to render
function Navigation(){
  return (
// Root component for navigation sections Nav
      <Navbar color="dark" dark expand="md">
         <NavbarBrand to="/" tag={Link}>Home Page</NavbarBrand>

         <ServicePanel/>

         <GuestPanel/>

      </Navbar>
   )
}

export default Navigation;
```

Registration.js – component that handles registration request to microservices and process response. To send a request and process response, we use *JavaScript ES6* stadard function called *fetch()*.

Usage Example:

```
….
// Function to handle submit button on the form
   handleSubmit(event) {

       let resStatus = 0;
// Step 1: compose a request
       fetch(regUrl, {
          //mode: 'no-cors',
          method: 'POST',
// Add headers
          headers: {
             'Accept': 'application/json',
             'Content-Type': 'application/json'
          },
// Compose request body. State is update as a form is being filled out, so by the time we click submit,
// all variables should be initialized. Convert this values into json string.
          body: JSON.stringify({
             email: this.state.email,
             password: this.state.password
          })
       })
// Step 2: Receive a response and conver into json. Additional values from header can be obtained as
// well at this point, for example response status code.
          .then(res => {
          resStatus = res.status;
          return res.json()
       })
// Step 3: Do something with json body from the response, for example updating state of the component
// with values from json. We also do a switch on response status to correctly handle different status
// code scenarios.
          .then(res => {
          switch (resStatus) {
             case 201:
                console.log(res.message);
                this.setState({status: res.message, registered: true});
                break;
             case 409:
                console.log(res.message);
                this.setState({status: res.message});
                break;
             case 500:
                console.log('server error, try again')
                this.setState({status: res.message});
                break;
             default:
                console.log('unhandled');
                this.setState({status: res.message});
                break;
          }});

       event.preventDefault();
   }
….
```

The examples give in this section of the chapter covers main aspects of developing with ReactJS and concepts are used in development of other components with similar functionalities.
Other components can be found project repository in react-web-client directory.

5.3 Database

5.3.1 Neo4j

The graph database – Neo4j is used to store user, role and verification token objects as nodes:

```
@NodeEntity
public class User implements UserDetails {
    @GraphId
    private Long id;
    @Index(unique = true)
    private String username;
    private String password;
    private boolean enabled;
    private boolean accountNonExpired;
    private boolean accountNonLocked;
    private boolean credentialsNonExpired;
    @Relationship(type = "HAS_ROLE",
            direction = Relationship.OUTGOING)
    private Set<Role> roles;
```

@NodeEntity: declare User node

@GraphId: similar to the primary key in MySQL

@Relationship: specify user has_role → role

Figure 47 neo4j model declare in spring project

```
@NodeEntity
public class Role {
    @GraphId
    private Long id;
    private UserRole name;
    @Relationship(type = "HAS_ROLE",
            direction = Relationship.INCOMING)
    private Set<User> users;
```

@NodeEntity: declare Role node

@GraphId: similar to the primary key in MySQL

@Relationship: specify role has_role ← user

```
@NodeEntity
public class VerificationToken {
    @Transient
    private static final int EXPIRATION =  60 * 24;
    @GraphId private Long id;
    private String token;
    private Date expiryDate;
    @Relationship(type = "VERIFY",
            direction = Relationship.OUTGOING)
    private User user;
```

@NodeEntity: declare Token node

@GraphId: similar to the primary key in MySQL

@Relationship: specify token verify → user

expiration: the valid time of token is 24 hours

All this annotations defines in Registration Service of the *Micro Service Me Up* project.

In the application.properities file add config setting to connext Neo4j database:

```
spring.data.neo4j.username=neo4j
spring.data.neo4j.password=1234
spring.data.neo4j.uri=http://34.217.77.79:7474  # this url is our online database, need to change.
```

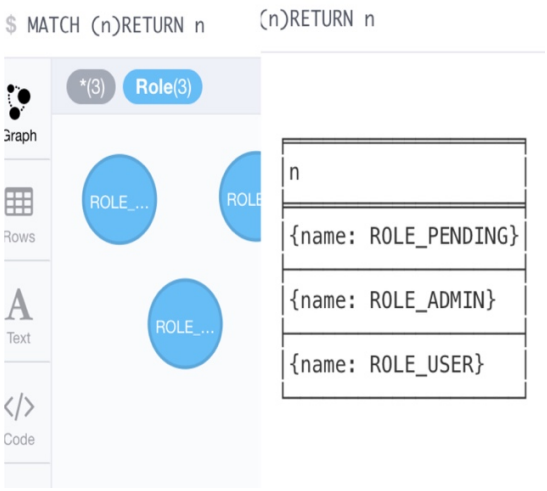


Figure 48 neo4j data store when reg setup

First, when the Registration Service sep up, three roles are created in the database with the help of `@PostConstruct` (The PostConstruct annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization) spring annotation.

Second, when a new user node created, a verify token node is created as well. Also two relationships, HAS_ROLE (user → pending_role) & VERIFY (user → token) is created.

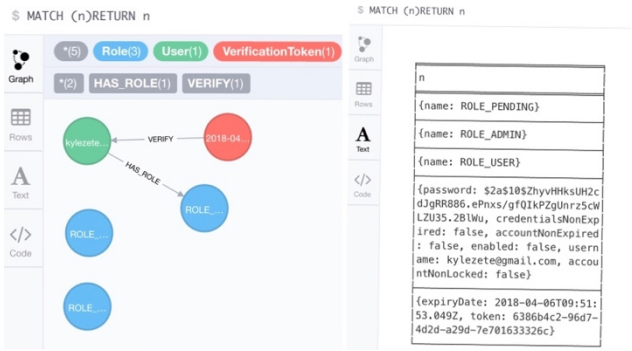


Figure 49 deo4j data store when token node added

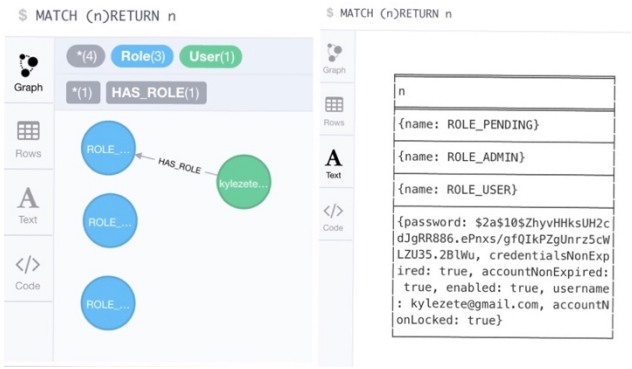


Figure 50 neo4j data store for user and role

Third, when such user is verified, the verify token node and the VERIFY (user → token) relationship will be deleted. And the HAS_ROLE relationship change from [user → pending_role] to [user → user_role].

5.3.2 MongoDB

We use MongoDB [10] to store user details and service items onjects used in Profile Service of the *Micro Service Me Up* project.

In the models of Profile Service, we define as follows:

```
@Id
private String id;

private String profileImg;
private String email;
private String nickName;
private String gender;
private String dob;
private String phoneNum;
private String aboutMe;
```

This Bean class contains basic info of a user. Note that there does not have password, because this is an independent model in Profile Service. Only valid user specified in the Registration Service (Neo4j) is able to have this extra user info.

Figure 51 mongo model declare in spring project

```
public class Item {
    @Id
    private String id;

    private String email;
    private String title;
    private String content;
    private String date;
```

This Item model describes a simple service meaage. Also it is working with the user which exists in the Neo4j database through Registration Server.

In the application.properities file add config setting to connext MongoDB database:

spring.data.mongodb.uri= mongodb://34.217.77.79:27017/cooldb

this url is our online database, need to change.

MongoDB is a document database, it stores data as a document (similar to JSON file).

```
> use cooldb
switched to db cooldb
> show collections
userDetail
item
```

In MongoDB database server, we create a database called cooldb manually. Inside this database, two collections (similar to table in MySQL) generated automatically when the Profile Service set up.

There are some examples of data stored in the MongoDB database:

Figure 52 mongo data store sample

5.4 Docker & AWS

Docker and Amazon Web Service are introduced in Technique Review part. In the *Micro Service Me Up* project, every microservice in the back-end, every clients and databases are dockerized in a container running on AWS responsively. There are some configuration for docker and AWS in both maven – spring boot and flask projects below:

5.4.1 Deploying Maven project with docker: [11]

- ✧ For different microservices, we need deploying to different ports to distinguish them.
  - Add " EXPOSE port " to DockerFile. For example: EXPOSE 8080.

- ✧ Dockerize the Maven project
  - Open Commend Line and direct to the project folder
  - Type *$ mvn package -Dmaven.test.skip=true docker:build*

    When this process finshed, a docker image will be created.

  *Note: Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Basic sample shows in Technique Review part.

- ✧ Build and run the docker image in Amazon Linux
  - Type:

    $ docker run -d --name NAME -e JAVA_TOOL_OPTIONS="-Dspring.profiles.active=aws -Deureka.instance.hostname=$HOSTNAME" --net="host" IMAGENAME

  *Note: Docker Hub (similar to github) can help you to move docker images to cloud.

5.4.2 Deploying Flask project with docker:

- ✧ Dockerize the Flask project need DockerFile like this:

```
FROM python:3.6
MAINTAINER Tanqed "tanqed@gmail.com"
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["app.py"]
```

- ✧ Dockerize the Flask project:
  - Open Commend Line and direct to the project folder
  - Type: *$ docker:build*

- ✧ Build and run the docker image in Amazon Linux
  - Typing normal run command:

    $ docker run --name NAME –p 5000:5000 IMAGENAME

5.4.3 Using Amazon Linux EC2 instances to run the docker and services.

(note: here skip docker installation in Amazon Linux)

In the Back-End servers of the *Micro Service Me Up* project, we need to specify publich IP to each microservices. Otherwise, Zuul (gataway) cannot communicate with every microservices. The reason is AWS using private IP for its EC2 instances with spring cloud by default. *** That is why we add the configuration class inside related microservices.**

*Also need to mention: it is necessary to add related port to the inbound rules in the security group of the Amazon Linux instance to enable the projecr can be accessed through the internet.

Chapter 6: System evaluation

6.1 Proove of concept

According to our aims and objectives stated in an introductory chapter of the dissertation, our primery goal was to gain an experience and insight of developing Cloud network of independent web services that are able to cooperate together across the network. We can state now that Microservices architecture bring handful of benefits to web development, although not without cost of effort put into it. Considering the number of various technologies used in development, it is hard for us to imagen how to manage all of this in one monolithic application.

Throughout our development process of microservices, we've been able to incrementaly add a new services to the system without affecting the workflow of already deployed services. The only exception would be a gateway service that had to be updated for every new type of service deployed. It was easy enough to add extra few lines to *configuration.yml* file and put gateway back online. Later we've discovered other techniques that could bring a continues integration into a gateway service, but will be discussed in recommendation part of the dissertation.

We didn't end up with a "spaghetti" code that is hard to read and debug. Each microservice is kept in its own project directory and source code is packaged in a standard manner for all of them. It is much easier to navigate across different service directories and find a specific functionality of your current interest.

Prehaps an important benefit of microservices architecture we've understood was the ability to freely change internal implementation of services, i.e., service layers and data layers in N-Tier architecture of an application. A bit different story is to change a view layers (rest controllers) as they represent an end-points of communication with other services and required certain degree of consistency, but this are the details of designing an overall orchestra of microservices that demand a precise understand of business requirements and what consumer expects as a return to request.

Regarding an automation of deploying services on dedicated Linux instances, we most definatly managed to speed up our development process. Docker technology freed us from monotonious work of preparing instances for each individual service, for example setting up an environment, downloading necessary frameworks and libraries. Docker also assisted us in a very fast deployment of databases we are using by pulling images for Neo4j and MongoDB from Dockerhub. As a price to pay for comfort Docker provides, we had to learn how to properly write a Docker files for deployment of Java Spring applications on AWS instances, as well as for Python Flask application.

6.2 Main known problems

6.2.1 Deploying a spring cloud (Netflix) project with AWS EC2 instance

One AWS EC2 instance has a public Ip and a private Ip address:

- ✧ A public IP address is an IPv4 address that's reachable from the Internet. You can use public addresses for communication between your instances and the Internet.
- ✧ A private IPv4 address is an IP address that's not reachable over the Internet. You can use private IPv4 addresses for communication between instances in the same network (EC2-Classic or a VPC).

We deploy Eureka Server & Client Services on AWS to build the cloud structure. Generally, Eureka Server discovered these services and provide communication ways (HostName/IpAddress).

By default, Eureka Server gets Client Services' private Ip. Then, a problem comes out:

One Services can connect to others **only** when all Services sets up on one EC2 instance or instances of one account.

To solve this problem, we need changing the private Ip to public Ip. The solution as follows:

In your Client Service project, creating a config class like this : [12]

```
@Configuration
@EnableAsync
@Import(InetUtils.class)
public class Configuration {
    @Value("${server.port}")     // get port from yml/properties file
    private int port;

    @Bean
    @Autowired
    public EurekaInstanceConfigBean eurekaInstanceConfig(InetUtils inetUtils) {
        EurekaInstanceConfigBean config = new EurekaInstanceConfigBean(inetUtils);
        AmazonInfo info = AmazonInfo.Builder.newBuilder().autoBuild("eureka");
        config.setHostname(info.get(AmazonInfo.MetaDataKey.publicHostname));
        config.setIpAddress(info.get(AmazonInfo.MetaDataKey.publicIpv4));
        config.setNonSecurePort(port);
        config.setDataCenterInfo(info);
        return config;
    }
}
```

Then, dockerize the project...

Note: the command for Image running should change to this: [13]

```
$ sudo docker run -d --name NAME -e JAVA_TOOL_OPTIONS="-Dspring.profiles.active=aws" --
net="host" IMAGENAME
```

6.2.2 Flask as externally visible server

When you run the server, you will notice that it can only be accessed from your own computer, and it cannot be accessed anywhere else on the network. This is because, by default, in debug mode, a user in the application can execute arbitrary Python code on your computer.

If you turn off debug or trust users on your network, you can make your server available externally by simply changing the call to method run() like this: [14]

```
app.run(host='0.0.0.0')
```

This lets your operating system listen to all public IPs.

6.2.3 Maven dependencies compatabilities

To get a quick set-up for the project on cloud, we used spring dependency bundles called spring-cloud-starter-parent that include dependencies for Netflix API, as well as Spring Boot starter itself. Cloud starters are coming in various versions like:

- Brixton
- Dalston
- Camden

For more details on versions, visit spring cloud.

The essence of potential problems with Spring Cloud lies in a question "What other technologies you plan to use in this particular project?". Each of Spring Cloud version available has it's own dependency build. Most of them depend on Spring 4.2.x, few on 4.3.x. This can cause a conflicts with newer technologies built for latest Spring releases. For example, if we want to use Neo4j database repositories, we have to run application on Spring 4.3.x version, otherwise repository won't be instantiated. Note that Mongodb can rely on 4.2.x.

To keep a consistency across microservices, we had to pick one spring-cloud-starter-parent and use it for all services to avoid possibilities of other compatibility problems.

6.2.2 Using Spring Cloud SSO outside Spring framework

One of the major issues that stopped our development of React client application was a difficulty of understanding a setup to use SSO (Single Sign-on) outside Spring framework. Spring Cloud provides a samples with implementation of SSO for clients that are hosted from Tomcat containers within Spring Boot applications. Complecated wiring of requests to http servlet filters and preparations for redirecting tokenized requests was beyond our current knowledge and understanding. AngularJS client also comes with initial setup to use SSO and uses its host container to send requests too. Attempting to adopt similar strategy with ReactJS didn't succeed and caused other problems with developing React application inside Spring Boot. Other thing to mention, it was uncomfortable to develop React together with Spring Boot, because it required to rebuild Maven solution with every change and rerun application to be able to see new results. It happened because Tomcat container was serving an older client bundle and didn't do hot reload. Consiquently we had to move React application away from Spring and do it in Node with benefits of hot reloading from built in Webpack watch configurations.

6.3 Proove of robustness

Unfortunatly we didn't have skills and experience in quickly deploying unit tests. In order to save time, we were focusing on sprint development of services and testing end-points with client mocking tools. Our main testing tool was Postman that acts like a client and able to send requests to services with various options. End-point were tested for successful acception of a request and responding with appropriate response, i.e., returning data on correct request or sending appropriate message if request was incorrect. We also tested services connectivity across different Linux instances to insure that Cloud network working and all requests and responses are going through gateway service Zuul.

6.3.1 Services comminication

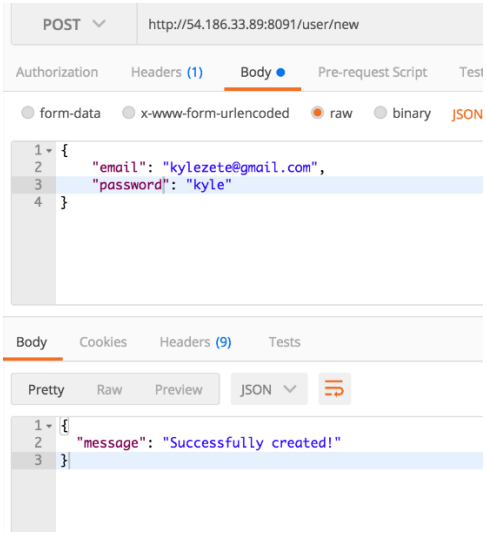


Figure 53 Tests: Successful Registration

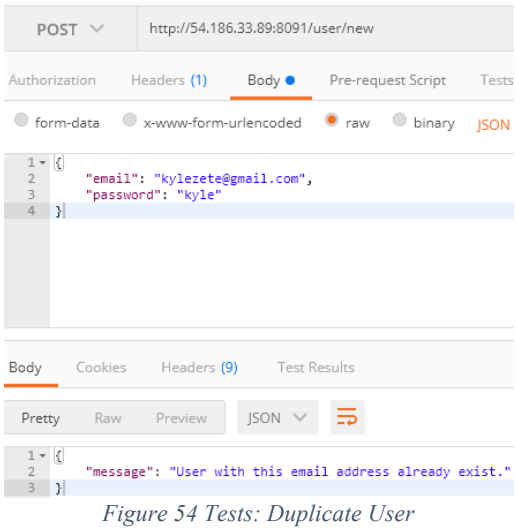


Figure 54 Tests: Duplicate User

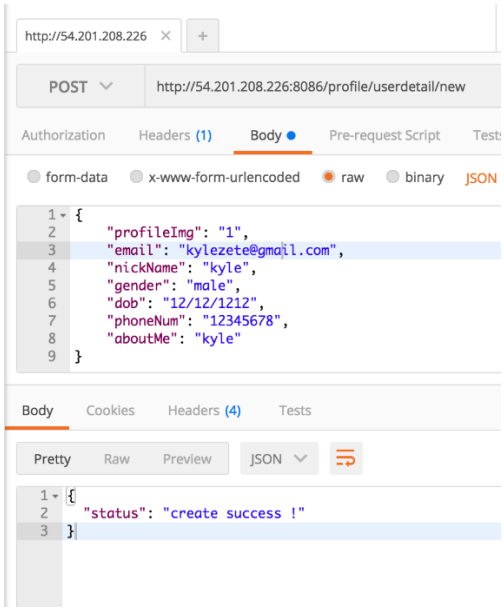


Figure 55 Tests: Successful Profile Update

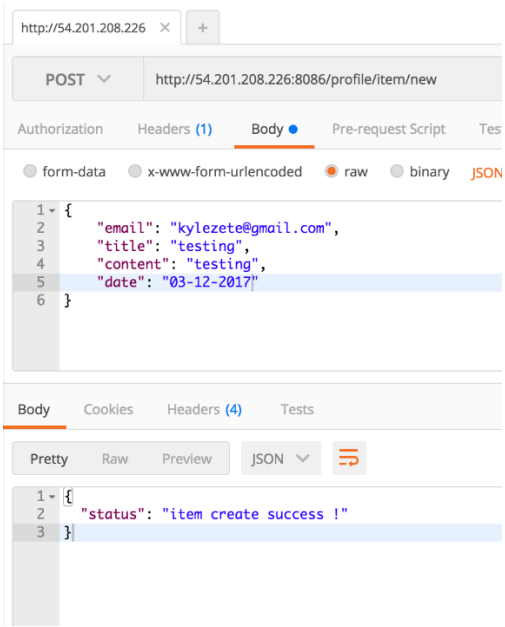


Figure 56Tests: Successfully Creating Dashpost

6.3.2 Cloud deployment

```
[ec2-user@ip-172-31-20-127 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
15b288089667        ed5b17255a90        "/bin/sh -c 'exec ..."   2 months ago        Up 2 months                             gateway
37830e183fd6        9780b968f5bb        "sh -c 'java $JAVA..."   2 months ago        Up 2 months                             discovery
[ec2-user@ip-172-31-20-127 ~]$
```

Figure 57 Cloud: Spring Cloud Infrastructure Containers

```
[ec2-user@ip-172-31-21-105 ~]$ docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS              PORTS               NAMES
c299d5630783        tanqed/registration    "sh -c 'java $JAVA..."   5 weeks ago         Up 5 weeks                              reg
3e1260047bd0        tanqed/auth            "sh -c 'java $JAVA..."   6 weeks ago         Up 3 weeks                              auth
c7fe7ede1677        6de238c5d58d           "sh -c 'java $JAVA..."   2 months ago        Up 2 months                             email
[ec2-user@ip-172-31-21-105 ~]$
```

Figure 58 Cloud: System Services Containers

```
[ec2-user@ip-172-31-20-229 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                                            NAMES
1eabd39cba58        mongo               "docker-entrypoint..."   8 days ago          Up 8 days           0.0.0.0:27017->27017/tcp                         mongo
39fd41662eb1        neo4j:3.0           "/docker-entrypoin..."   8 days ago          Up 2 minutes        7473/tcp, 7687/tcp, 0.0.0.0:7474->7474/tcp       neo4j
```

Figure 59 Cloud: Databases

```
[ec2-user@ip-172-31-27-97 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
cfd0526f3741        tanqed/flaskreg     "python app.py"          6 weeks ago         Up 6 weeks          0.0.0.0:5000->5000/tcp   flask
66e589dda551        tanqed/profile      "sh -c 'java $JAVA..."   7 weeks ago         Up 7 weeks          0.0.0.0:8092->8092/tcp   profile
```

Figure 60 Cloud: Profile & Flask Services

Chapter 7: Conclusion/Recommendations

In this chapter we would like to tell a reader about our overall impressions and feeling about developing Cloud services, remind about our project aims and objectives, recapping each of them to draw a conclusion line for our research. We also would like to give recommendations for future development iterations of Micro Service Me Up project.

7.1 Research and development experience

It was an enjoyable experience researching and developing microservices for both of us. Overall outcome of the project left us satisfied. The amount of problems we encountered during a research definatly increased our abilities to find solutions quickly and adapt them to fit current sprints. On our way we met various programming and engineering problems, making us think not only about software production, but also networking, frameworks and libraries versions in use, and database schemas. What we are trying to say is, our research produced learning outcomes from many fields of Computer Engineering and Software development.

We also learned a lot from working in team. We have covered each others weaknesses and were able to shine with our strengths to create pleasant and productive environment to work in. Even thou we had few tense moments of disagreement during development process, for example discussing how security works in microservices, we managed to resolve situations gracefully, leaving both of us happy. We learned that it is important to listen to both sides, backup your ideas and intensions, and try to make discussion rather than an argument. If for some reason it didn't go that way, our best approach was to leave it for a bit, go outside for fresh air and relax, then come back with peace in mind and resolve situation.

We learned about planning out steps in advance to help schedule realistic goals within certain timeframe and to see broader picture of the project itself. To increase amount of work done, we often split responsibilities and worked parallel, but pairing up to solve difficult problems or while conducting research.

7.2 How to continue development of Micro Service Me Up

Considering the size of the Micro Service Me Up project, there is plenty of room to continue development, improving and extending functionalities. In this paragraph we will mention what we didn't finish, what could be done differently and what it is possible to do in future.

To start with one of the most difficult and important topics in developing Cloud software is security. Using an existing frameworks like Spring Cloud Security and OAuth2 is a steep learning curve and time consuming. We still weren't able to secure access to other services by means of tokens and user roles. If we were to continue development and given enough time, we would set it as high priority goal.
Same would go for React client which is not dependant on existing implementation of Spring Cloud Security and has to be developed by means of using other libraries that integrate AOuth2 tokens, or write your own way.

We've managed to put foundation for user hierarchy in database schema and assign various roles and authorities to users. It can be used to restirict operational methods to a specific role and authorities. For example you can implement administration service that is exclusively used by users with an administrational role. Further more you can assign authorities to each administrator to reduce access to some methods,i.e., God Admin would be able to delete user, but moderator would only be able to suspend.

Custom emailing package that is used in registration service to make compose request to email service can be packaged as an individual *.jar* file and used as the dependency across other microservices that would require such functionality.

Profiles and dashboards functionalities in profile service should be decoupled into separate microservices. MongoDB database may allow them to share it, as different documents are created and no relationships maintained. Replication and shrading techniques can help load balancing.

One new technology we recommend to add into back-end stack is Kafka Stream API. You can set up servers that implement queues that broadcast output streams to its subscribers. One good example we can think of using it is to keep databases in synch if replication is required. Stream consumer service of some kind can be used as a subscriber, consume a stream once notified and update all databases related to the data in the stream. Production of input streams can be handle with services that filters client's request data redirected by gateway, perform database consistency checks and send it out to Kafka servers to broadcast stream of data. Kafka API is highly configurable, for example you can set broadcast TTL of certain stream to 2 days after which it will be removed to save memory.

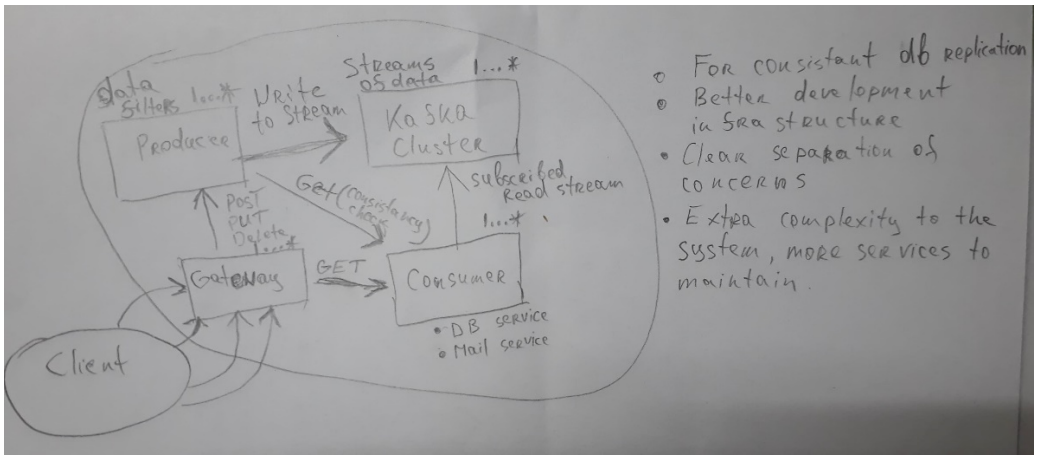


Figure 61 Kafka Integration Plan

As for developing React client application, it requires refactoring and better understanding of development principals for modern React applications, e.g., redesign application into presentational and smart components, learn how to use local browser storage to create container components for better data control. Libraries like Redux can be used to create single data access point within application used by smart components to feed presentation or update the container. With Redux most of the React components can be functional, without the need to hold a state, resulting in better code readability and ease to debug. Angular mobile client should also be refactored into scripts that executing only one type of functionality to improve code readability.

Ideally, it is better to develop one adaptive web applications for both pc and mobile devices. We demonstrated that it can be done with 2 different libraries and which to use for developing is matter of taste. What could be done in next development iterations is to develop a cross platform mobile application that shares the back-end with web application. Options you have here would be Ionic/AngularJS/TypeScript or React Native library. We would like to note that if to choose ReactJS for both applications, a lot of coded components can be shared by both projects, making it faster to develop. Once existing microservices are polished and well tested, it would be a time to begin using Docker Swarm for even distribution of software across multiple linux instance(nodes) and effectively fast services replication for load balancing. Docker Swarm can be used to scale up/down services in real time from command line of the manager node instance.

Tip for updating gateway service config file – there are few possible ways to update files inside running Docker containers from CLI, it can be handy in maintaining continues itegration of new types of services. There is no need to stop gateway container to update its config for routing to the services.

7.3 Aims and objectives recap

- **Research in the area of Cloud web development and Microservices - □**

  Research has been conducted and documented. Software is produced as a prove of concept and work.
- **Gain practical experience in using trending software development methodologies that are suitable for Microservices development. □**

  Software was developed under Agile methodology processes. Techniques like Scrum and Pair Programming were used throughout development.
- **Gain an insight and a better understanding the various technologies that are associated with development of Cloud based web services. □**

  Produced hands-on experience in Cloud technologies like Java Spring Cloud, AWS Console, Docker, RESTful Services, web clients with AngularJS and ReactJS, Cloud databases Neo4j and MongoDB.
- **Designing an appropriate architecture for Cloud service. □**

  Microservices Architecture was used to combine various technologies for advanced web development on Cloud.

- **Undertake comprehensive research in full stack development, including database management, back-end and front-end development. □**

  Software was successfully produced and stable enough to continue development.
- **Finding solutions for problems encountered throughout development cycle. □**

  Problems are recorder and documented.
- **Research in automation technologies to speed up development of services. □**

  Docker was extensively used to improve development processes and achieve production speed.
- **Evaluate design and application throughout development iterations. □**

  Software design was constantly improving thought iterations, gaining in size and refactored to meet industrial standards as close as possible. Development process was recorder on Github as a commits log. Necessary testing were performed to meet basic requirements to pass software on to next iteration.
- **Draw a conclusion on developing Microservices, evaluating benefits and drawbacks of using architecture, and set recommendations for new or further development. □**

  Conclusion is documented and recommendations are given on how to continue developing Micro Service Me Up project.

Thank for your attention,
Tangqi Feng & Eduards Vagals,
TanqEd Team.

Chapter 8: Bibliography/References

[1] S. Newman, “O'Reilly Media,” 2 2015. [Online]. Available: http://shop.oreilly.com/product/0636920033158.do. [Accessed 11 11 2017].

[2] T. Killalea, “ The hidden dividends of microservices,” *Communications of the ACM,* vol. 59, no. 8, pp. 42-45, 8 8 2016.

[3] A. Ciuffoletti, “Automated Deployment of a Microservice-based Monitoring Infrastructure,” *Procedia Computer Science,* vol. 68, pp. 163-172, 2015.

[4] spting_cloud_community. [Online]. Available: https://cloud.spring.io/spring-cloud-netflix/. [Accessed 20 9 2017].

[5] Tushar_Gupta, 3 12 2014. [Online]. Available: https://www.codeproject.com/Articles/845939/AngularJs-Quick-Start. [Accessed 1 4 2018].

[6] Neo4j. [Online]. Available: https://neo4j.com/product/. [Accessed 2 4 2018].

[7] MongoDB. [Online]. Available: http://www.runoob.com/mongodb/mongodb-intro.html. [Accessed 2 4 2018].

[8] tutorialspoint-sendingEmail. [Online]. Available: https://www.tutorialspoint.com/java/java_sending_email.htm. [Accessed 18 11 2017].

[9] spring-cloud-samples, 15 1 2018. [Online]. Available: https://github.com/spring-cloud-samples/authserver.

[10] tutorialspoint-mongo. [Online]. Available: https://www.tutorialspoint.com/mongodb/index.htm. [Accessed 20 1 2018].

[11] spotify. [Online]. Available: https://github.com/spotify/docker-maven-plugin. [Accessed 10 10 2017].

[12] W. Rożkow, 27 8 2015. [Online]. Available: https://github.com/spring-cloud/spring-cloud-netflix/issues/432. [Accessed 10 10 2017].

[13] Rob, “stackoverflow,” 6 8 2015. [Online]. Available: https://stackoverflow.com/questions/31779344/how-should-i-configure-spring-cloud-with-netflix-zuul-and-eureka-in-a-docker-con. [Accessed 10 11 2017].

[14] Flask. [Online]. Available: http://www.pythondoc.com/flask/quickstart.html#id2. [Accessed 5 4 2018].

Chapter 9: Appendices

This chapter is related to software installation instructions and how to set up development ground.

GitHub: https://github.com/TanqEdPublic/Micro-Service-Me-Up
Demo: https://youtu.be/8Ow0CRWHyyU

9.1 Local Development Site Instructions

Back-end installation requirements

Java 8 – programming language to develop, compile and run back-end microservices.

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Development tools installation requirements

Maven – package manager and versioning control tool for Java platform applications. Installs necessary project dependencies via POM.xml configuration file. Builds Java executable jars.

https://maven.apache.org/install.html

Docker – automation tool for independent application deployment within isolated Linux instances inside Docker containers. Installs necessary application dependencies through Docker file or using Maven package manager where applicable.

https://docs.docker.com/install/

Git – project versioning control and collaborative tool for developing iterative projects within a group or solo. https://git-scm.com/downloads

Maven packaging – microservices applications packaging commands.
Run the following command in CLI within project root directory where POM.xml is located:

```
mvn package -Dmaven.test.skip=true docker:build
```

NB: Maven creates Docker image for projects with Docker plugin in POM.xml

Read our manual in Wikis: https://github.com/TanqEdPublic/Micro-Service-Me-Up/wiki/Docker

Docker deployment – running image in local environment differs from Cloud.

```
docker run --name=ProjectName -p 8080:8080 -t ImageName
```

Front-end

AngularJS + Spring – everything is done through Maven packaging and Docker deployment.
Instructions above.

ReactJS + libraries – requirements to develop and run ReactJS client application
First need to install NodeJS - https://nodejs.org/en/

Setting up React base

```
npm install -g create-react-app
create-react-app my-app-name
```

Downloading dependency libraries

```
npm install react-router –save
npm install react-router-dom –save
npm install bootstrap –save
npm install jquery –save
npm install popper.js –save
npm install reactstrap –save
```

Run application

```
npm start
```

Python – install Python 3+ for developing and running TanqEd OpenID platform.
https://www.python.org/downloads/

Python Flask + libraries for local developmen

```
pip3 install Flask
pip3 install requests
```

Deploying with Docker will install necessary Python libraries through requirements.txt file.
Run normal Docker command to create an image

```
docker build -t my-python-flask-image:lates .
```

NB: don’t forget ‘.’ , a dot, indicating Docker file is in current directory.

Docker deployment

Creating containers from images

```
docker run –name=ContainerName -p 8080:8080 -t ImageName
```

NB: automatically will start the container

Start container manually

```
docker start ContainerName
```

Stop container

```
docker stop ContainerName
```

Enter console of the running container

```
docker attach ContainerName
```

9.2 Cloud Development Site Instructions

These instructions are related to software installation on Linux Centos instances hosted by AWS.

Back-end / Front-end development tools

NB – no libraries installation required on Cloud. Docker will handle those requirements.

Docker – installing Docker on Linux Centos

https://docs.docker.com/install/linux/docker-ce/centos/

Docker command sequency to push images to Dockerhub

1. Create a copy of an image for desired Dockerhub account

```
docker tag discovery tanqed/discovery:latest
```

2. Push image to Dockerhub

```
docker push tanqed/discovery:latest
```

Docker command to pull images from Dockerhub

```
docker pull tanqed/discovery:latest
```

Docker command to create and start a new Java Spring application container in AWS

```
docker run -d –name ContainerName -e JAVA_TOOL_OPTIONS="-
Dspring.profiles.active=aws" --new="host"
```

Maven – https://maven.apache.org/install.html

Same usage as in local development environment

Git – https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Same usage as in local development environment