**PLP Case Study Implementation Documentation**

**Description:**

There are four functional components for which we have designed independent micro services and integrated as per the constraints. Individually they are independent and collectively they are performing one action by fetching details from others. As they are individual micro services even if one needs to be changed, others will not be effected.

Three of the functional components are back-end services and the final one acts as a front-end controller which connects those three functionalities.

**Functional Components:**

**Back-end services:**

**Employee Code Module**: This micro service focuses on employee details for expense management. The operations defined in this module are add, delete, update and view the employee details.

**Project Code Module**: This micro service focuses on project details for expense management. There is a user who can add, delete, update and view the project details.

**Expense Code Module**: This micro service focuses on expense details for expense management. There is a user who can add, delete, update and view the expense details.

**Front-end services:**

**Expense Claim Details Module**: This micro service acts as **front-end** controller. The functionality of this micro service is to integrate the back-end micro services i.e. Employee Code Module, Project Code Module and Expense Code Module. The portal is created through which a user can claim, view, update and delete the expense.

**Micro Service Connectivity:**

Service Registration

When you have multiple processes working together they need to find each other.The developers at Netflix had this problem when building their systems and created a registration server called Eureka. Fortunately for us, they made their discovery server open-source and Spring has incorporated into Spring Cloud, making it even easier to run up a Eureka server. Here is the complete discovery-server application:

@SpringBootApplication

@EnableEurekaServer

public class ServiceRegistrationServer {

public static void main(String[] args) {

SpringApplication.run(ServiceRegistrationServer.class, args);

}

}

Spring Cloud is built on Spring Boot and utilizes parent and starter POMs. The important parts of the [POM](https://github.com/paulc4/microservices-demo/blob/master/pom.xml) are:



By default Spring Boot applications look for an application.properties



By default Eureka runs on port 8761, but here we will use port 1111 instead. Also by including the registration code in my process it might be a server or a client. The configuration specifies it is a client and stops the server process trying to register with itself.

# Creating a Microservice: ExpenseCode

# A microservice is a stand-alone process that handles a well-defined requirement.

# What makes it special is that it registers itself with the discovery-server at start-up. Here is the Spring

# To register to the EurekaServer we must make our microservice as a client and respective dependency must be added in pom.xml

# 

# Boot startup class:

# 

The annotations do the work:

1. @EnableAutoConfiguration - defines this as a Spring Boot application.
2. @EnableDiscoveryClient - this enables service registration and discovery. In this case, this process registers itself with the *discovery-server* service using its application name (see below).



Note that this file

1. Sets the application name as ExpenseCode. This service registers under this name and can also be accessed by this name - see below.
2. Specifies a custom port to listen on (2222). All my processes are using Tomcat, they can’t all listen on port 8080.
3. The URL of the Eureka Service process - from the previous section.

Run the ExpenseApplication now and let it finish initializing. Refresh the dashboard [http://localhost:1111](http://localhost:1111/) and you should see the EXPENSECODE listed under Applications.



# Accessing the Microservice: ExpenseDetail

# To consume a RESTful service, Spring provides the RestTemplate class. This allows you to send HTTP requests to a RESTful server and fetch data in a number of formats - such as JSON and XML.

## Encapsulating Microservice Access

Here is part of the EmployeeDetail for my client application:



Accesing other microservices should be impelemented in Service Layer but for simplicity here we have implemented in Controller itself.



**Pom.xml :**

*Same as the ExpenseCode*

*Docker Containerization*

1. Create a EC2 Instance.
2. Open Putty Terminal and connect to the instance using IP address and respective Key.
3. Enter ‘sudo su’ to get root privileges
4. Install Docker ‘yum install docker’
5. Add docker to the User group ‘usermod -a -G docker ec2-user’
6. Download docker-compose file
   1. curl -L https://github.com/docker/compose/releases/download/1.9.0/docker-compose-`uname -s`-`uname -m` | sudo tee /usr/local/bin/docker-compose > /dev/null
   2. chmod +x /usr/local/bin/docker-compose
7. Start the docker service ‘service docker start’
8. ‘chkconfig docker on’
9. Create a directory and put all the necessary files there.
   1. mkdir ExpenseMangament
   2. cd ExpenseMangament
   3. Copy all the .jar files of the microservices ( To build a jar file first go to the folder of the project in cmd and type ‘mvn clean install’
   4. Create Dockerfile for each microservice and name it as ‘Dockerfile-{servicename}’ in the same directory.

Enter the following code inside the file.

‘vi Dockerfile-EurekaServer’

--------------------------------------------------

FROM openjdk:8-jdk-alpine

VOLUME /tmp

ADD EurekaServer.jar EurekaServer.jar

EXPOSE 1111

ENTRYPOINT ["java","-jar", "EurekaServer.jar"]

*Similarly for every Service we need to make a Dockerfile*

1. Create a docker-compose.yml file
   1. ‘vi docker-compose.yml’



NB: Dockerfile name should be relevant as made earlier for each service

1. To run the docker-compose type ‘/usr/local/bin/docker-compose up’

Reference:

https://github.com/rasingha/CloudBatch\_ExpenseManagmentSystem/