Technische Universität München

Cloud Computing Course

WS - 2017

Exercise - 3

“Building Microservices”

Authors:
Prof. M. Gerndt,
Prof. S. Benedict,
Anshul Jindal

Table of Contents

I. Introduction

1.1 Purpose of this document

This document provides guidance and material to assist the relevant person, in understanding the Microservice architecture and building them. This includes following:

- What is Microservice?
- Understanding Seneca.js
- Building Microservices
- Deployment of Microservices

1.2 Prerequisites

Following requirements are mandatory:

- Account on LRZ [1]

Following requirements are recommended (not mandatory):

- Basic knowledge of Linux console commands.
- Knowledge of Node.js

1.3 Background

1.3.1 Monolithic Architecture

When we develop a server-side application, we start with a layered application which usually have following components:

1. A database (which consist of many tables usually in a relational database management system).
2. A client-side user interface (consisting of HTML pages and/or JavaScript running in a browser)
3. A server-side application.

But despite having the layered architecture the application is packaged into a single monolith (a massive structure) type architecture and deployed on the cloud. The deployed structure will look like as shown in below figure.

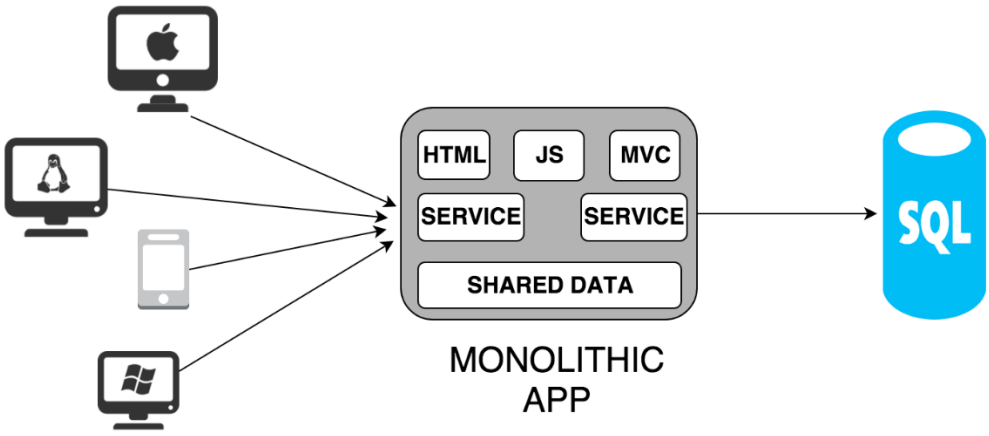

Figure 1: Monoloithic Architecture
(https://www.slashroot.in/difference-between-monolithic-and-microservices-based-architecture)

This type of deployment architecture is called as monolithic architecture. Industries are using this approach since a long time to develop enterprise applications. Many companies have invested years to build enterprise application with monolithic approach. Sometimes it is also called multi-tier architecture because monolithic applications are divided in three or more layers or tires. The main benefits for deploying using such architecture are:

1. **Easy to develop**: The biggest advantage is that it is simple and easy to develop.
2. **Simple to test:** We can implement end-to-end testing by simply launching the application and testing the UI with desired outputs.
3. **Simple to deploy**: We just have to copy the packaged application to a server for the deployment.
4. **Easy Horizontal Scaling**: Simple to scale horizontally by running multiple copies of the complete application behind a load balancer.

Similarly, there are many other advantages of the monolithic architecture. In the early stages of the projects this architecture works well and basically most of the big and successful applications which exist today. But after some time, problems start coming. A few of them to list down are:

1. **Limitation in size and complexity**: This approach has limitation in size and the complexity. There cannot be simply just continuous addition to previous code.
2. **Too large and complex:** Newly joining developers have to understand the complete application. But the application is too large and complex to fully understand and make changes fast and correctly.
3. **Slow start time:** The application start time is directly dependent on the size of the application. If the application size grows the start time of the application also increasers.
4. **Re-Deployment of the Application on Updates**: If there were any updates, then the entire application must be re-deployed which increases the application downtime.
5. **Difficult in Scaling:** Scaling monolithic applications can also be difficult when different modules have conflicting resource requirements.
6. **Reliability:** Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.
7. **Barrier to adopting new technologies**: Since changes in frameworks or languages will affect an entire application it is extremely expensive in both time and cost.

1.3.2 Microservice Architecture

To avoid the monolithic architecture drawbacks, the idea here is to split the application into a set of smaller, interconnected services. Each service intercommunicates with a common communication protocol like REST web service with JSON. Each service run individually either in single machine or different machine, but they execute its own separate process. Each service may **have own database or storage system**, or they can share common database or storage system. Microservice is all about distribute or break application in small chunks. The deployed microservice architecture structure will look like as shown in below figure.

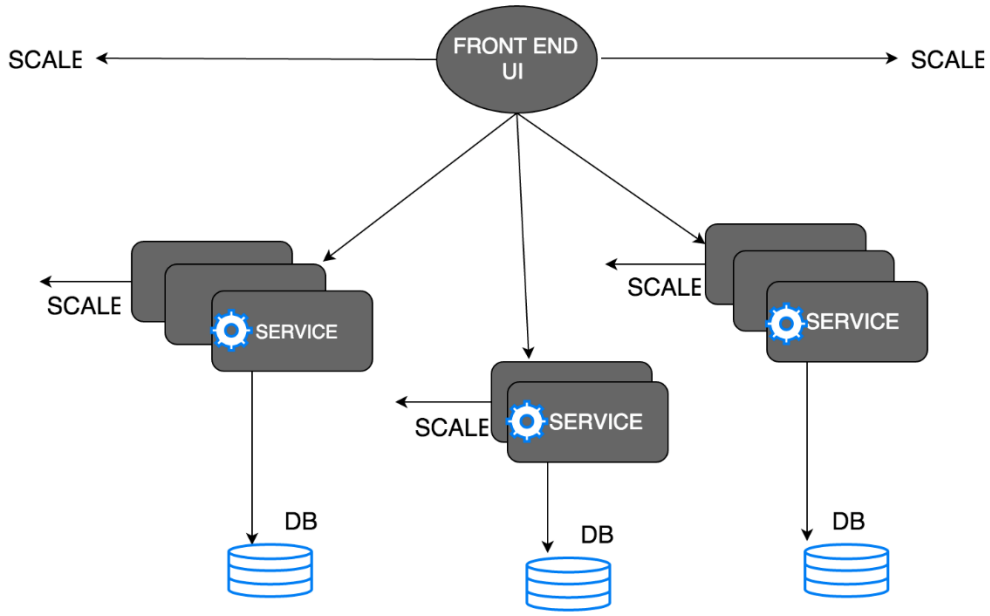

Figure 2: Microservice Architecture
(https://www.slashroot.in/difference-between-monolithic-and-microservices-based-architecture)

As shown in the above diagram, the front end is completely separated from the core logic and can be scaled independently. Also, the backend components are separated to different services, with individual databases. Having individual databases for each of the service can help in modifying database schemas without impacting any other component in the infrastructure. Each service shown above can be placed behind their own load balancers, to achieve more throughput and availability of that service. Another main advantage of using services is that each service can use their own required database types. Some services can use NoSQL databases if required, some can use the traditional RDB. The main benefits for deploying using such architecture are:

1. **Easier to understand and maintain:** Here the application is decomposed into a set of manageable services which are much faster to develop, and much easier to understand and maintain.

2. **Independence of Service:** Each service can be developed independently by a team that is focused particularly on that service.

3. **No Barrier on Adopting New Technologies:** Here the developers are free to choose whatever technologies make sense for their service and not bounded to the choices made at the start of the project.

4. **Independent Service Deployment:** Microservice architecture enables each microservice to be deployed independently. As a result, it makes continuous deployment possible for complex applications.

5. **Easy Scaling:** Microservice architecture enables each service to be scaled independently.

Although microservice architecture comes with a lot of benefits but it has its own drawbacks also. To list down a few of them are:

1. **Complexity of creating a distributed system:** Developers must deal with the additional complexity of creating a distributed system.
    - Developer tools/IDEs are oriented on building monolithic applications and don't provide explicit support for developing distributed applications.
    - Testing is more difficult
    - We need to choose and implement an inter-process communication mechanism based on either messaging or RPC and write code to handle partial failure and take into account other fallacies of distributed computing.
    - Implementing use cases that span multiple services without using distributed transactions is difficult
    - It is more difficult to implement changes that span multiple services. In a monolithic application you could simply change the corresponding modules, integrate the changes, and deploy them in one go. In a Microservice architecture you need to carefully plan and coordinate the rollout of changes to each of the services.

2. **Deployment complexity:** Deploying a microservices-based application is also more complex. A monolithic application is simply deployed on a set of identical servers behind a load balancer. In contrast, a microservice application typically consists of many services. Each service will have multiple runtime instances. And each instance need to be configured, deployed, scaled, and monitored. In addition, you will also need to implement a **service discovery mechanism**[1]. Manual approaches to operations cannot scale to this level of complexity and successful deployment a microservices application requires a high level of automation.

1.3.3 What is an API Gateway?

Suppose we are developing a mobile client shopping application, it's likely that we need to implement a product details page, which displays information about any given product.

Usually the product details page displays a lot of information. Like:

1. Price
2. Name

[1] https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

3. Description
4. Order history
5. Customer Reviews
6. Recommendation
7. Others

When using a monolithic application architecture, a mobile client would retrieve this data by making a single REST call to the server application. A load balancer routes the request to one of N identical application instances. The application would then query various database tables and return the response to the client.

In contrast, when using the microservices architecture the data displayed on the product details page is divided among multiple microservices. So, the mobile client must query those services but the mobile client does not know which endpoints to query. Below figure illustrates this.

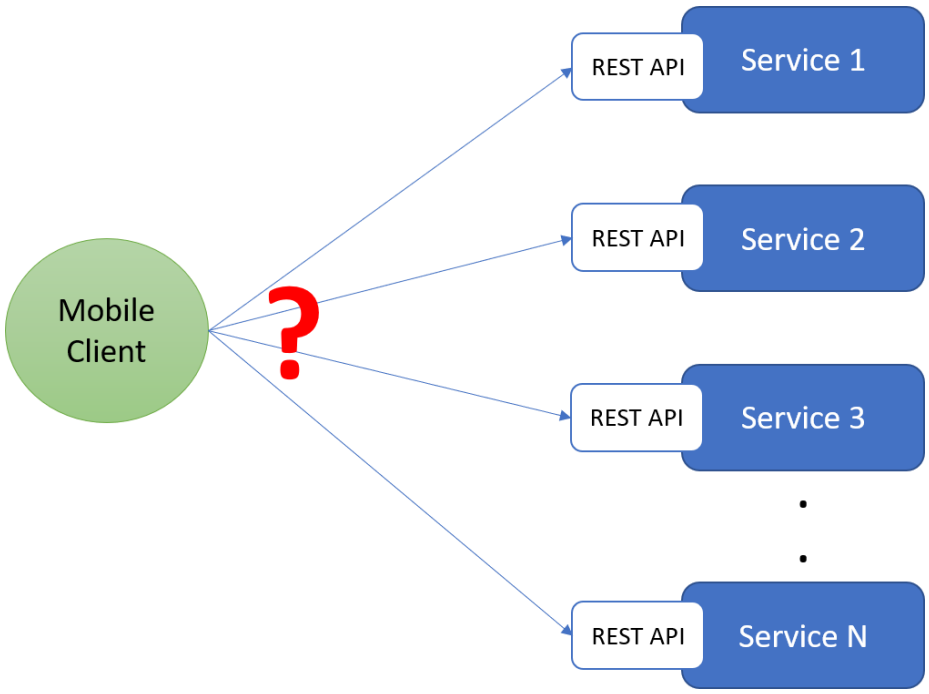

Figure 3: API Endpoints to query

In theory, a client could make requests to each of the microservices directly. Each microservice would have a public endpoint. This URL would map to the microservice's load balancer, which distributes requests across the available instances.

Unfortunately, there are some problems. One problem is the mismatch between the needs of the client and the fine-grained APIs exposed by each of the microservices. The client in this example has to make 7 separate requests. In more complex applications it might have to make many more. While a client could make that many requests over a LAN, it would probably be too inefficient over the public Internet and would be impractical over a mobile network. This approach also makes the client code much more complex. Another problem with the client directly calling the microservices

is that some might use protocols that are not web-friendly. One service might use Thrift binary RPC while another service might use the AMQP messaging protocol. Neither protocol is particularly browser- or firewall-friendly and is best used internally. An application should use protocols such as HTTP and WebSocket outside of the firewall.

Another drawback with this approach is that it makes it difficult to refactor the microservices. Over time we might want to change how the system is partitioned into services. For example, we might merge two services or split a service into two or more services. If, however, clients communicate directly with the services, then performing this kind of refactoring can be extremely difficult.

A much better approach is to use what is known as an **API Gateway[2].** An API Gateway is a server that is the single-entry point into the system. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.

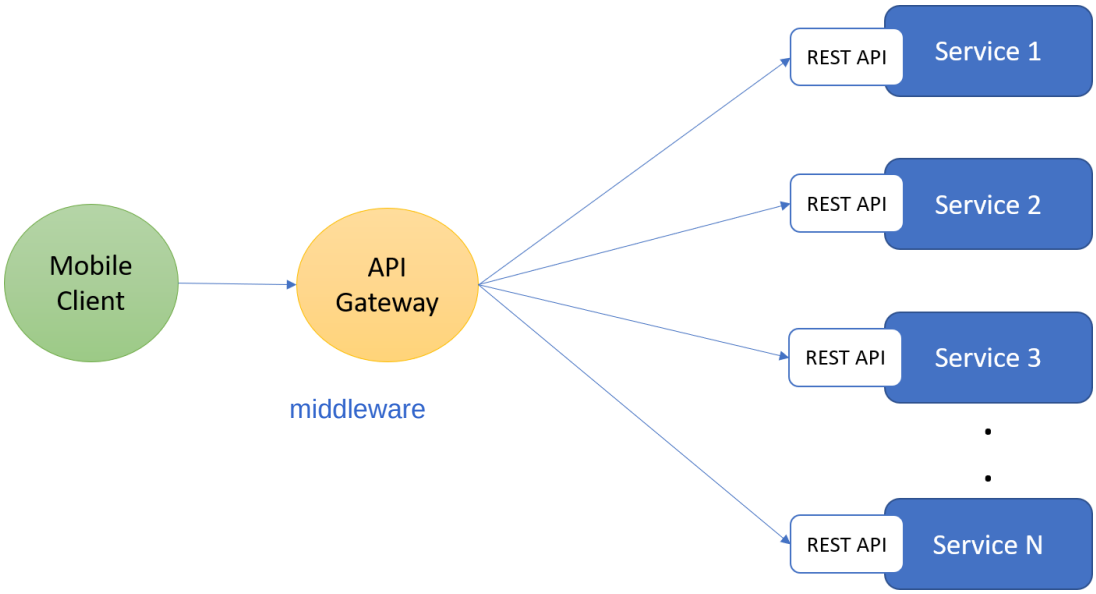

Figure 4: API Gateway

The API Gateway is responsible for request routing, composition, and protocol translation. All requests from clients first go through the API Gateway. It then routes requests to the appropriate microservice. The API Gateway will often handle a request by invoking multiple microservices and aggregating the results. It can translate between web protocols such as HTTP and Web Socket and web-unfriendly protocols that are used internally.

1.3.4 What is Service Discovery?

Suppose you are writing the code for microservice architecture application and there are some services present in it. One of the module invokes a service that has a REST API. To make a request, your code needs to know the network location (IP address and port) of a service instance. In a

[2] http://microservices.io/patterns/apigateway.html

traditional application running on physical hardware, the network locations of service instances are relatively static. For example, your code can read the network locations from a configuration file that is occasionally updated. However, in a microservices application this is not the case as shown in the following diagram

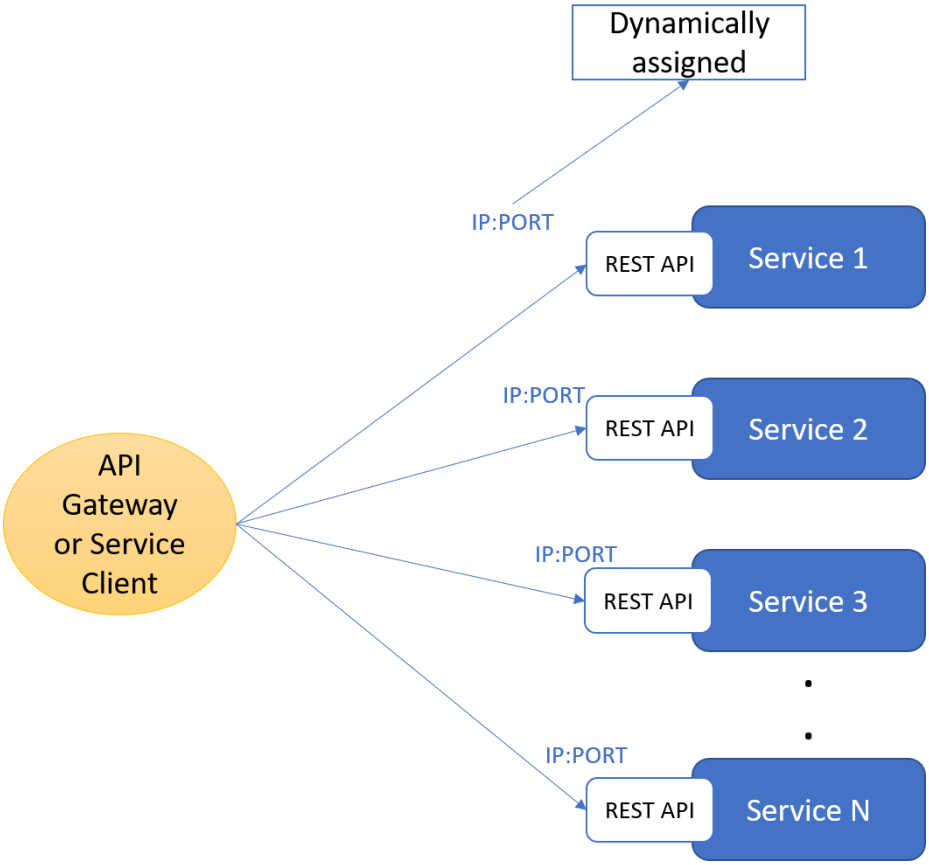

Figure 5: Dynamically Assigned Network Configuration to Services

client must query those services but the mobile client does not know which endpoints to query. Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of Autoscaling, failures, and upgrades. There are two main service discovery patterns: client-side discovery and server-side discovery. Let's first look at client-side discovery.

Client-Side Discovery: When making a request to a service, the client obtains the location of a service instance by querying a Service Registry[3], which knows the locations of all service instances. Shown in below diagram:

[3]Database of services, their instances and their locations. For More info

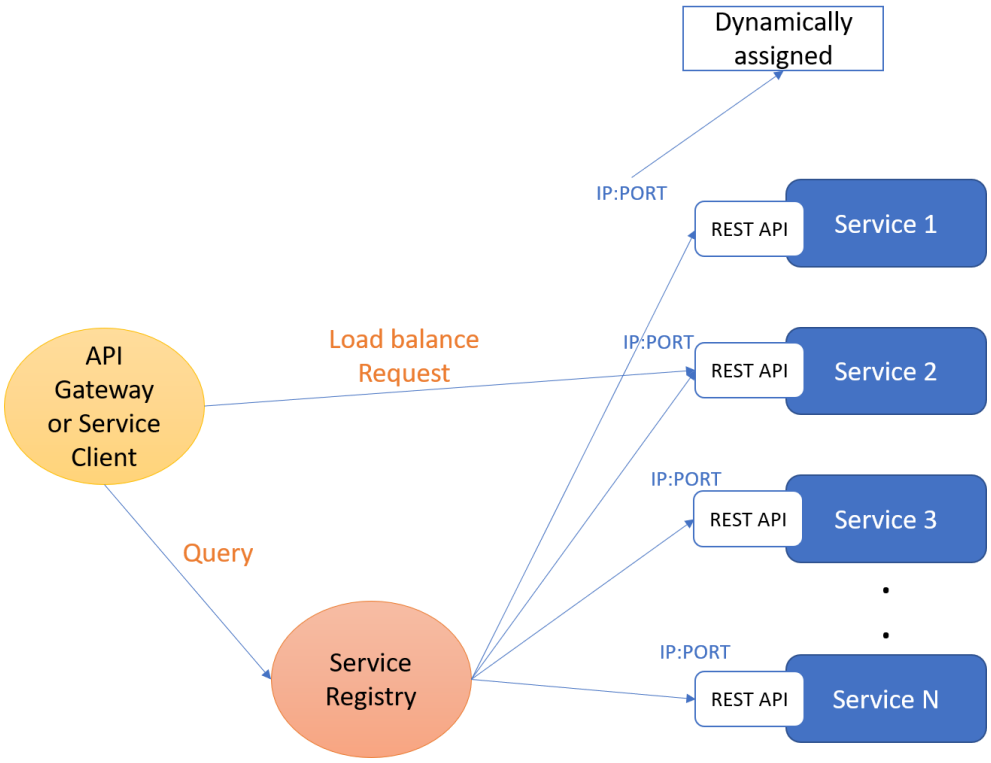

Figure 6: Client-Side Service Discovery

Server-Side Discovery: When making a request to a service, the client makes a request via a router (a.k.a load balancer) that runs at a well-known location. The router queries a service registry, which might be built into the router, and forwards the request to an available service instance. As shown in below diagram:

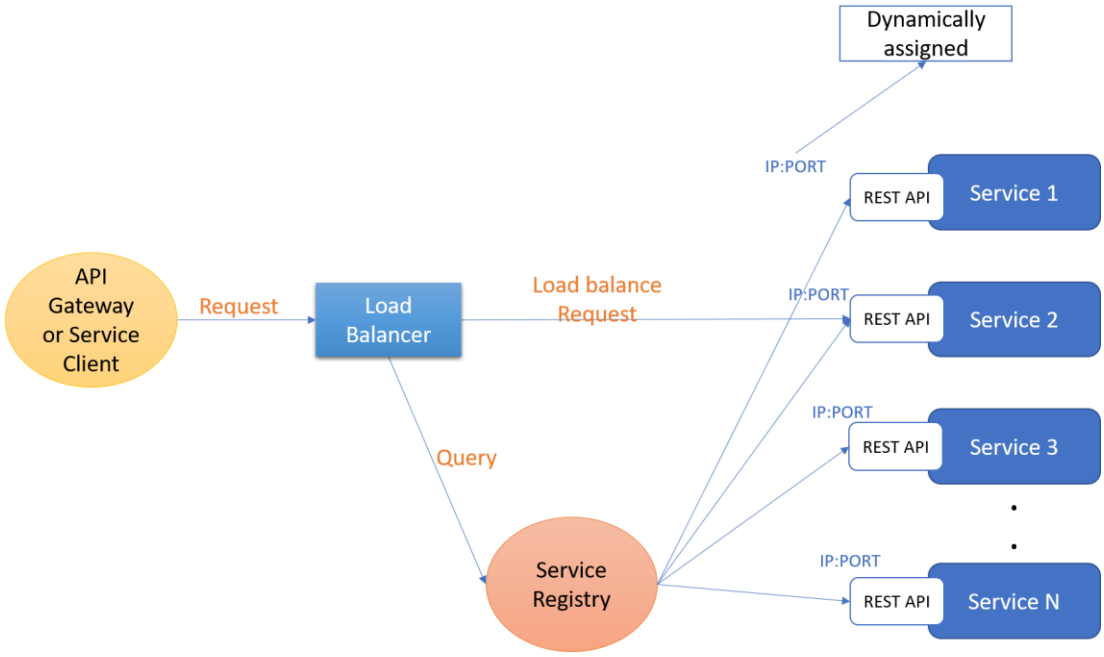

Figure 7: Server-Side Service Discovery

1.3.5 Seneca.js[4]

A microservices framework for building scalable applications in Node.js. Seneca provides features that make it simple to build microservices, and clients to use them, along with many plugins to extend the function of your microservice. Seneca lets you build message based microservice systems with ease. You don't need to know where the other services are located, how many of them there are, or what they do. Everything external to your business logic - such as databases, caches and third-party integrations - is likewise hidden behind microservices.

This decoupling makes your system easy to continuously build and change. It works because Seneca has the following three core features:

- **Pattern matching:** Instead of fragile service discovery, you just let the world know what sort of messages you care about.
- **Transport independence:** You can send messages between services in many ways, all hidden from your business logic.
- **Componentisation:** Functionality is expressed as a set of plugins which can be composed together as microservices.

Messages are JSON objects. They can have any internal structure you like. Messages can be sent via HTTP/S, TCP, message queues, publish/subscribe services or any mechanism that moves bits around. From your perspective as the writer of a service, you just send messages out into the world. You don't need to know which services receive them.

Then there are the messages you'd like to receive. You specify the property patterns that you care about, and Seneca (with a little configuration help) makes sure that you get any messages sent by other services that match those patterns. The patterns are very simple: just a list of key-value pairs that must match the top-level properties of the JSON message.

Let's start with some code. We will create two microservices, one that will return with the hello message and another that makes use of it.

```
var seneca = require('seneca')()
seneca.add('role:api,cmd:hello', (msg, reply) => {
 reply(null, {answer: "Hello "+ (msg.name)})
})
seneca.act({role: 'api', cmd: 'hello', name: 'CCS'}, function (err, result) {
 if (err) return console.error(err)
 console.log(result)
})
```

For the moment, this is all happening in the same process, without network traffic. In-process function calls are a type of message transport too!

[4] This section is mostly from senecajs.org. For more Info Click here

The **seneca.add** method adds a new action pattern to the Seneca instance. It has two parameters:

- **pattern:** the property pattern to match in any JSON messages that the Seneca instance receives.
- **action:** the function to execute when a pattern matches a message.

The action function has two parameters:

- **msg:** the matching inbound message (provided as a plain object).
- **respond:** a callback function that you use to provide a response to the message.

The respond function is a callback with the standard error, result signature.
Let's put this all together again:

```
seneca.add({role: 'api', cmd: 'hello'}, function (msg, respond) {
 var result = "Hello "+ msg.name
 respond(null, {answer: result })
})
```

In the sample code, the action adds the string "Hello" to the name provided via the **name** property of the message object. Not all messages generate a result, but as this is the most common case, Seneca allows you to provide the result via a callback function.

In summary, the action pattern **role:api,cmd:hello** acts on this message:

```
{role: 'api', cmd: 'hello', name: 'CCS'}
```

to produce this result:

```
{answer: Hello CCS}
```

There is nothing special about the properties **role** and **cmd**. They just happen to be the ones you are using for pattern matching.

The **seneca.act** method submits a message to act on. It has two parameters:

- **msg**: the message object.
- **response_callback**: a function that receives the message response, if any.

The response callback is a function you provide with the standard error, result signature. If there is a problem (say, the message matches no patterns), then the first argument is an Error object. If

everything goes according to plan, the second argument is the result object. In the sample code, these arguments are simply printed to the console:

```
seneca.act({role: 'api', cmd: 'hello', name: "CCS"}, function (err, result) {
  if (err) return console.error(err)
  console.log(result)
})
```

II. Exercise

The Goal of the exercise is to build 3 microservices, out of which two will be already provided to get used to the code and architecture but the third one you need to develop. The three microservices are:

1. **HelloWorld Service:** This service responds back with the welcome message to the user who is trying to access this service.
2. **Product Description Service:** This Service will respond back with the product description (URL and the product Name) based upon the product Id provided as the query parameter.
3. **Product Price Service:** This Service will respond back with the product price based upon the product Id provided.

2.1 What will be Used?

1. Node.js
2. Docker
3. Seneca.js
4. Express
5. Bluebird

2.2 Code Directory Structure

To start with it, let's see first the directory structure of the code provided:

```
    docker-compose.yml

├──hello-world-service
        .dockerignore
        Dockerfile
        helloWorld.js
        index.js
        package.json

├──product-descp-service
        .dockerignore
        Dockerfile
        index.js
        MOCK_DATA.json
        package.json
        product_descp.js

├──product-price-service
        .dockerignore
        Dockerfile
        index.js
        MOCK_DATA.json
        package.json
        product_price.js

└──server
```

```
        .dockerignore
        app.js
        Dockerfile
        index.js
        package.json

├──config
        index.js

└──services
        helloWorld.js
        productDescp.js
        productPrice.js
```

Following section will cover the short explanation of the each file:

2.2.1 docker-compose.yml[5]

Compose is a tool for defining and running multi-container Docker applications. This file is used for creating microservices architecture and manage them together. With this file we can specify what all services we have and from where these need to be built.

```yaml
version: '2'
services:
  server:
    build: ./server
    image: HUB_ID/microservice:server
    ports:
      - "8080:8080"
  hello-world-service:
    build: ./hello-world-service
    image: HUB_ID/microservice:hello
  product-descp-service:
    build: ./product-descp-service
    image: HUB_ID/microservice:productdescp
  product-price-service:
    build: ./product-price-service
    image: HUB_ID/microservice:productprice
```

Figure 8: Docker Compose File

2.2.2 hello-world-service

This is a microservice to respond back with the welcome message to the user. It is created using Seneca. Following are sub parts of it:

[5] For more Information click here

2.2.2.1 helloWorld.js

This file describes the actual business logic of this service. Based upon the input pattern the function is called. Here sayWelcome function is called if the pattern **{role:helloWorkd, cmd:Welcome}** is matched.

```
module.exports = function (options) {
    //Add the patterns and their corresponding functions
    this.add('role:helloWorld,cmd:Welcome', sayWelcome);

    //Describe the logic inside the function
    function sayWelcome(msg, respond) {
        if(msg.name){
            var res = "Welcome "+msg.name;
            respond(null, { result: res });
        }
        else {
            respond(null, { result: ''});
        }
    }
}
```

Figure 9:helloWorld.js

2.2.2.2 index.js

This file is like the main function of a program. Here the require of necessary modules (Seneca) and import the plugin created in the above file are done along with the port on which the service need to be run. This file is the start point for the service.

```
require('seneca')()
    .use('helloWorld')
    .listen({ port: 9001 });
```

Figure 10:index.js

2.2.2.3 package.json

This is similar to node package.json file where all the dependencies are listed along with various commands to performed.

```
{
  "name": "helloWorld",
  "version": "1.0.0",
  "description": "Hello World Service",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "keywords": [],
  "author": "Anshul Jindal (ansjin)",
  "license": "ISC",
  "dependencies": {
    "seneca": "^3.2.2"
  }
}
```

Figure 11:package.json

2.2.2.4 Dockerfile

This file is used to make the container for this service and run it inside the container.

```
FROM alpine:latest

RUN apk update && apk add nodejs && rm -rf /var/cache/apk/*

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

COPY . /usr/src/app

RUN npm install

CMD ["npm", "start"]
```

Figure 12:Dockerfile

2.2.3 product-descp-service

This service has the similar structure as the above one, except that here a MOCK_DATA.json file is provided which consists of the products fake data. This service, based upon the productid return the product name and the URL. This service is not fully completed, your Task will be to complete this service in the same manner as the above service.

2.2.3.1 product_descp.js

This file describes the actual business logic of this service. Based upon the input pattern the functions are called. This file is Incomplete. You need to complete it.

2.2.3.2 index.js

This file is like the main function of a program. Here the require of necessary modules (Seneca) and import the plugin created in the above file are done along with the port on which the service need to be run. This file is the start point for the service.

```
require('seneca')()
   .use('product_descp')
   .listen({ port: 9002 });
```

Figure 13:index.js

2.2.3.3 package.json

This is similar to node package.json file where all the dependencies are listed along with various commands to performed.

2.2.3.4 Dockerfile

This file is used to make the container for this service and run it inside the container.

2.2.3.5 MOCK_DATA.json

This file contains the fake data of the products.


```
[
  {
    "product_id": 1,
    "product_name": "Daltfresh",
    "product_url": "https:\/\/theglobeandmail.com\/
    "product_company": "Tagtune",
    "product_price": 42,
    "product_count": 10,
    "product_number_of_users_liked": 636
  }
```


Figure 14: MOCK_DATA.json

2.2.4 product-price-service:

This is the service which will return price of the product based upon the product Id. This service is not fully completed, your Task will be to complete this service in the same manner as the other services.

2.2.5 Server:

This is the main service of the application which interacts with the user and the other services. It's task is to invoke the service and return the result back to the user. This holds the same structure as the other node.js application with the express framework used. The only difference comes when we want to invoke a service based upon the api end point. The service invoke is shown in below figure:

```
router.route('/exercise3')
    .get(function(req, res)
    {
        join(
            helloWorldService.sayWelcome(req.query.name),
            productDescpService.getProductURL(req.query.productId),
            productDescpService.getProductName(req.query.productId),
            productPriceService.getProductPrice(req.query.productId),
            function (resulthelloWorld, productDescpServiceURL, productDescpServiceName,productPriceServicePrice )

                var ex3_response_message = {
                    "hello": resulthelloWorld.result,
                    "product_id": req.query.productId,
                    "productURL": productDescpServiceURL.result,
                    "productPrice": productPriceServicePrice.result,
                    "productName": productDescpServiceName.result
                };
                res.send(ex3_response_message);
            }
```

Here we have route **"/exercises/exercise3**" which is handled by Express.

Here bluebird's *join* is used for concurrent service calls. As we can see different service calls are done and once they are completed, their output in the combined form is sent back to the user.

The services directory contains the configuration and link the service based upon the called function from the app.js file. helloWorld.js file looks like this:

```
/**
 * import the seneca package
 */
const seneca = require('seneca')();
const Promise = require('bluebird');
const config = require('../config');
/**
 * Convert act to Promise
 */
const act = Promise.promisify(seneca.client({ host: config.helloWorld_service.host, port:

/**
 * Service Method
 */
const SAY_WELCOME = { role: 'helloWorld', cmd: 'Welcome' };

/**
 * Call Service Method
 */
const sayWelcome = (name) => {
    return act(Object.assign({}, SAY_WELCOME, { name }));
};

module.exports = {
    sayWelcome
};
```

Here the host and port information is specified for the service where it is actually running. Then the service method is defined and call to those service methods.

The **product_descp.js** and **product_price.js** are incomplete. You need to complete it in similar fashion as above.

2.3 Running the Application

To run the application, you need to first install docker and docker compose. Please check previous exercise to know how to install docker. To install docker-compose run the following commands:

1. curl -L https://github.com/docker/compose/releases/download/1.13.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
2. chmod +x /usr/local/bin/docker-compose

Once docker-compose is installed, go into the root directory of the application and run the following command:

```
docker-compose up
```

If you need to build again, run this command:

```
docker-compose up --build
```

This command will use docker-compose.yml file and build all the services and put them in different containers.

2.4 Testing the Application

Run the api on the browser, the api format will look something like this (productid can be changed):

http://YOUR_VM_IP:8080/exercises/exercise3?name=CCS&productId=3

and the output will contain a below message with all the fields values set according to the product id.

```
{
    "hello": "",
    "product_id": "",
    "productURL": "",
    "productPrice": "",
    "productName": ""
};
```

2.5 Tasks to be completed

As part of the exercise3 complete the following tasks:

1. Complete the microservice **product-description-service(to get product name and URL)** and **product-price-service(to get product price),** based upon the product id passed as the query parameter. You need to complete the following files
    a. **product-price-service/product_price.js**
    b. **server/services/product_price.js**
    c. **product-descp-service/product_descp.js**
    d. **server/services/product_descp.js**
    e. **docker-compose.yml (add your docker hub id)**
2. Install docker and docker compose on the VM.
3. After installation run this application on the VIM as explained above. Check all the services running using **docker ps** command and testing on the browser.
4. Enable docker remote API:
    a. Edit the file /lib/systemd/system/docker.service
    b. Modify the line that starts with ExecStart to look like this
        ExecStart=/usr/bin/docker daemon -H fd:// -H tcp://0.0.0.0:4243
        Where the addition is "-H tcp://0.0.0.0:4243"
    c. Save the modified file
    d. Run systemctl daemon-reload
    e. Run sudo service docker restart
    f. Test that the Docker API is indeed accessible:
        curl http://localhost:4243/version

g. Enable port **4243** on your VM so that docker API can be accessed from outside network using your VM IP.

III. Result Delivery

You already know how this works 😊. To submit your application results you need to follow this:

1. Open the cloud Class server url
2. Login with your username and password.
3. After logging in, you will find the button for exercise3
4. Click on it and a form will come up where you must provide your VM ip on which your application is running.

   Example:

   10.0.23.1
5. Then click submit.

Important points to Note:

1. Make sure your VM and your application is running after following all the steps mentioned in this manual.
2. We will grade you based upon the number of exercises completed by you.
3. You will get to see, what your application has submitted to the server.
4. You can submit as many times until the deadline of exercise.
5. Multiple submission will overwrite the previous results.

2 References

[1] "LRZ," [Online]. Available: https://www.lrz.de/english/.

[2] "DockerWiki," Docker, [Online]. Available: https://en.wikipedia.org/wiki/Docker_(software).

[3] "DockerParts," [Online]. Available: (https://cdn-images-1.medium.com/max/800/1*K7p9dzD9zHuKEMgAcbSLPQ.png).

[4] "DockerHub," [Online]. Available: https://hub.docker.com/.

[5] "Docker Installation," Docker, [Online]. Available: https://docs.docker.com/engine/installation/.

[6] "Docker tag," Docker, [Online]. Available: https://docs.docker.com/edge/engine/reference/commandline/tag/.