

Cloud Computing – WS 2017

Exercise 3 : Building Microservices

5th December 2017

Anshul Jindal

anshul.jindal@tum.de

Index

- Exercise 2 Solution
- Exercise 3: Introduction
- Exercise 3: To Develop Microservice Architecture
- Exercise 3: Code Structure
- Exercise 3: Code Explanation
- Tasks To be Completed
- Submission

Exercise 2 Solution

1. Add an API in your application:

1./exercises/exercise2 : Which sends a message "group 'GroupNumber' application deployed using docker".

```
/**
 * Exercise 2: Send a message "group 'GroupNumber' application
 deployed using docker"
 * Replace GroupNumber with your group number
 * This api will be called from the server
 */
router.route('/exercise2')
  .get(function(req, res)
  {
    res.send("group 1 application deployed using docker");
  });
```

Just to replace GroupNumber with your group
number(here replaced with 1)

Docker File

```
FROM node:alpine

# Create app directory
RUN mkdir -p /usr/src/client
WORKDIR /usr/src/client

# Install app dependencies
COPY package.json /usr/src/client/
RUN npm install
# Bundle app source
COPY . /usr/src/client

EXPOSE 8080

CMD [ "node", "clientServer.js"]
```

Use a Docker base Image

This image is based on the popular [Alpine Linux project](#), available in [the alpine official image](#). Alpine Linux is much smaller than most distribution base images (~5MB), and thus leads to much slimmer images in general.

Create Application Directoy and set it as working one.

Install Application dependencies using package.json

Copy Source code

Export a port number

Write the start command

```
docker build -t "account_id"/"application_image_name"
```

3. Create your docker hub account and push your application image to it by following the procedure stated in detail document. **Name of your application image should be as `cloudcomputinggroup'GroupNumber'`**
4. Start a VM and pull this application image on it and run it using docker (you should pull and run the image otherwise exercise would not be able to pass successfully).

Already shown both as demo in the last exercise class

```
docker push "your_account_id"/"your_application_image_name"
```

```
docker run -p 8080:8080 "your_account_id"/"your_application_image_name"
```

5. Enable docker remote API

[Unit]

```
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
After=network.target docker.socket firewalld.service
Requires=docker.socket
```

[Service]

```
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
EnvironmentFile=-/etc/default/docker
ExecStart=/usr/bin/dockerd -H fd:// $DOCKER_OPTS -H tcp://0.0.0.0:4243
ExecReload=/bin/kill -s HUP $MAINPID
LimitNOFILE=1048576
# Having non-zero Limit*s causes performance problems due to accounting overhead
# in the kernel. We recommend using cgroups to do container-local accounting.
```

Edit the file /lib/systemd/system/docker.service

Add the following highlighted line

Save and run systemctl daemon-reload

Run sudo service docker restart

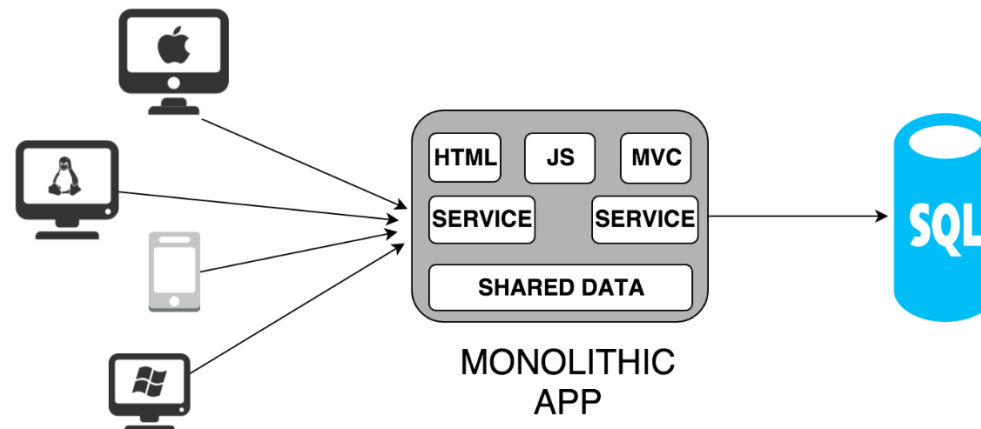
Enable port 4243 in iptables

Exercise 3

Exercise 3 : Building Microservices

Introduction: Monolithic Architecture

- Here a typical server-side application have a layered structure with following components:
 - A database (which consist of many tables usually in a relational database management system).
 - A client-side user interface (consisting of HTML pages and/or JavaScript running in a browser)
 - A server-side application.



Introduction: Benefits of Monolithic Architecture

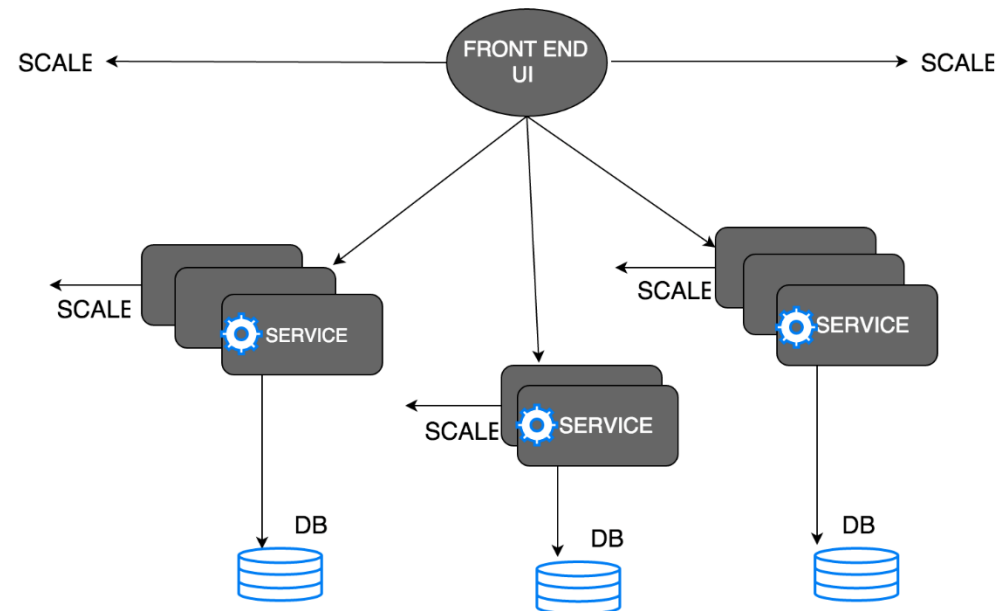
- **Easy to develop:** The biggest advantage is that it is simple and easy to develop as everything is handled at one place.
- **Simple to test:** We can implement end-to-end testing by simply launching the application and testing the UI with desired outputs.
- **Simple to deploy:** We just have to copy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

Introduction: Drawbacks of Monolithic Architecture

- Limitation in size and complexity: There cannot be simply just continuous addition to previous code.
- Too large and complex
- Slow start time
- Re-Deployment of the Complete Application on Updates
- Difficult in Scaling
- Reliability
- Barrier to adopting new technologies

Introduction: Microservices Architecture

- 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.
 - run individually either in single machine or different machine, but they execute its own separate process.
 - may **have own database or storage system**, or they can share common database or storage system.



Introduction: Benefits of Microservices Architecture

- Easier to understand and maintain
- Independence of Service
- No Barrier on Adopting New Technologies
- Independent Service Deployment
- Easy Scaling

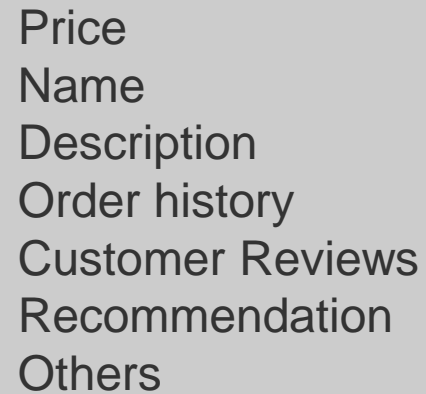
Introduction: Drawbacks of Microservices Architecture

- **Complexity of creating a distributed system**
 - Developer tools/IDEs are oriented on building monolithic applications
 - Testing is more difficult
 - We need to choose and implement an inter-process communication mechanism
 - Implementing use cases that span multiple services without using distributed transactions is difficult
- **Deployment complexity:**
 - 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.
 - Need to implement a **service discovery mechanism**.

Introduction: What is an API Gateway ?

Suppose we are developing a mobile client shopping application

And the product details page displays the following information of the product:



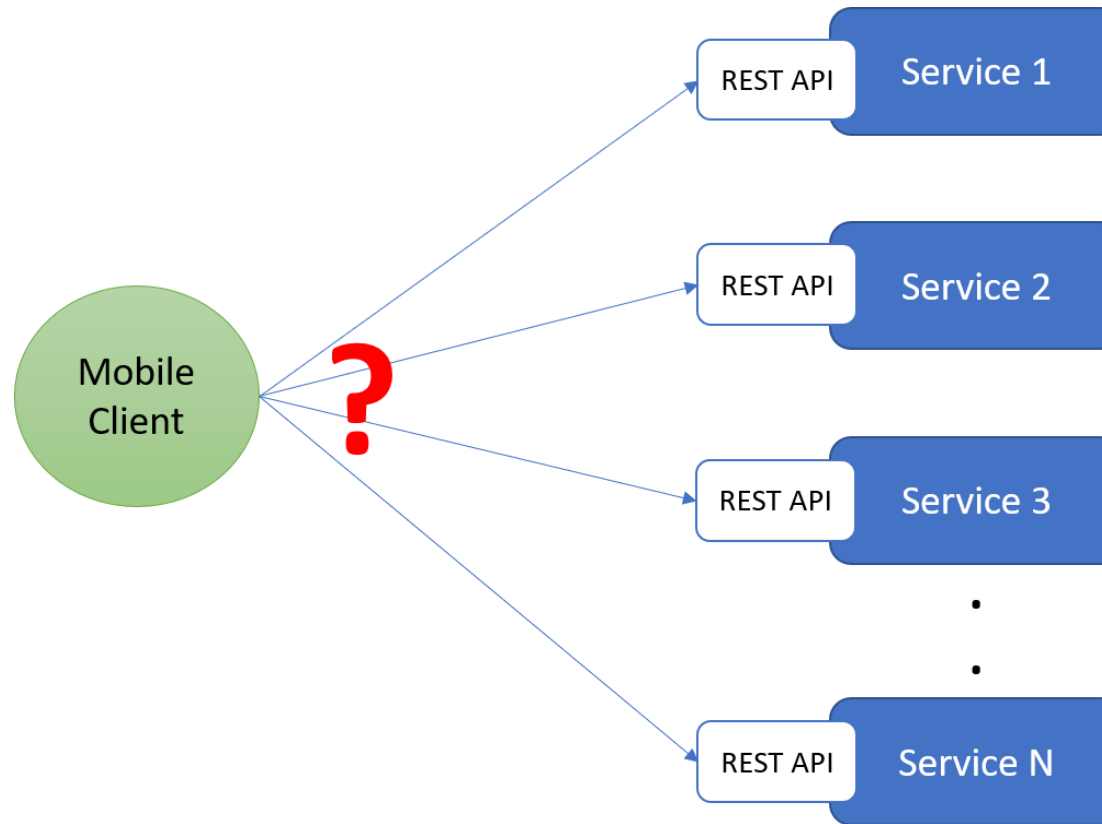
- Price
- Name
- Description
- Order history
- Customer Reviews
- Recommendation
- Others

Monolithic Application: Make a single REST call to the server application ->

A load balancer routes the request to one of N identical application instances ->
application then query database tables and return the response to the client.

Introduction: What is an API Gateway ?

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.



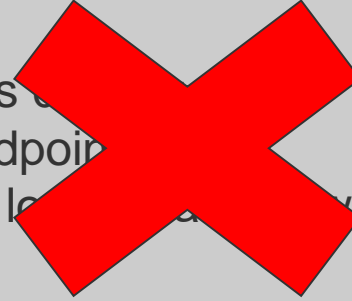
Introduction: What is an API Gateway ? Cont..

A Possible Solution:

client requests to each of the microservices

Each microservice would have a public endpoint

This URL would map to the microservice's load balancer which distributes requests across the available instances.



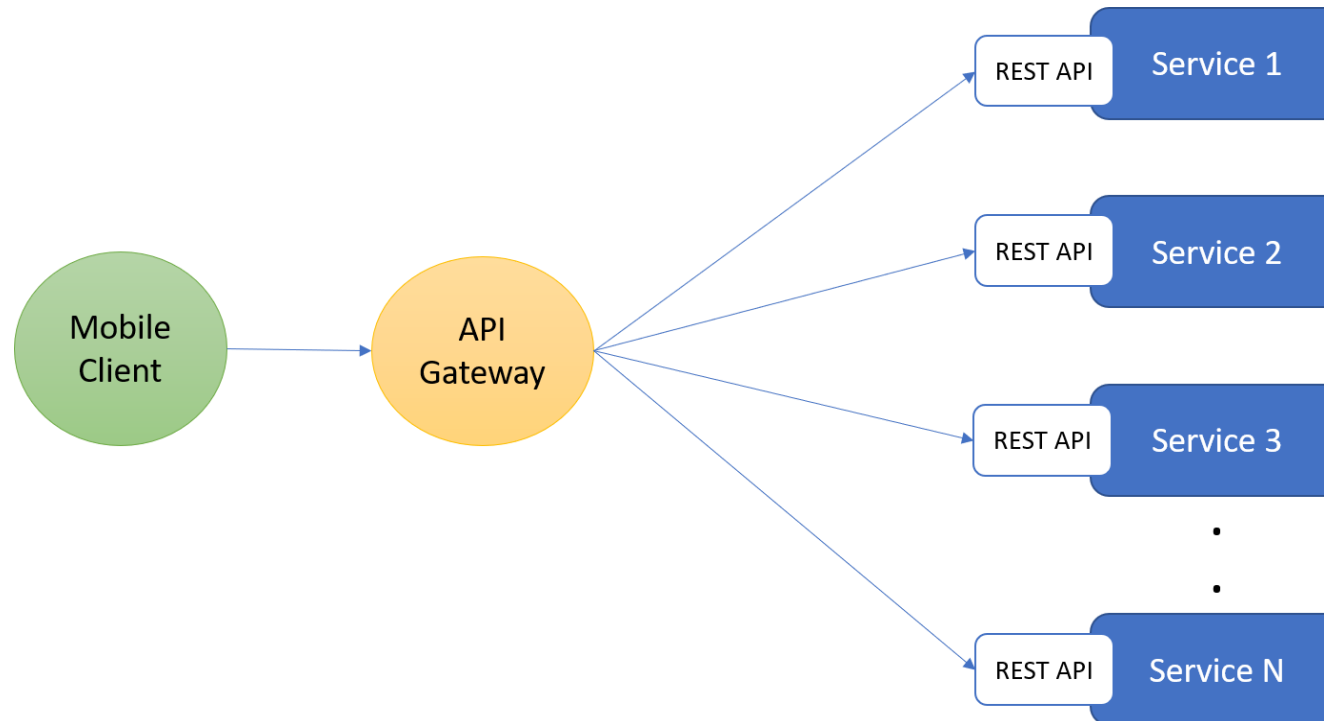
Why no?

- 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.
- difficult to refactor the microservices

Introduction: What is an API Gateway ? Cont..

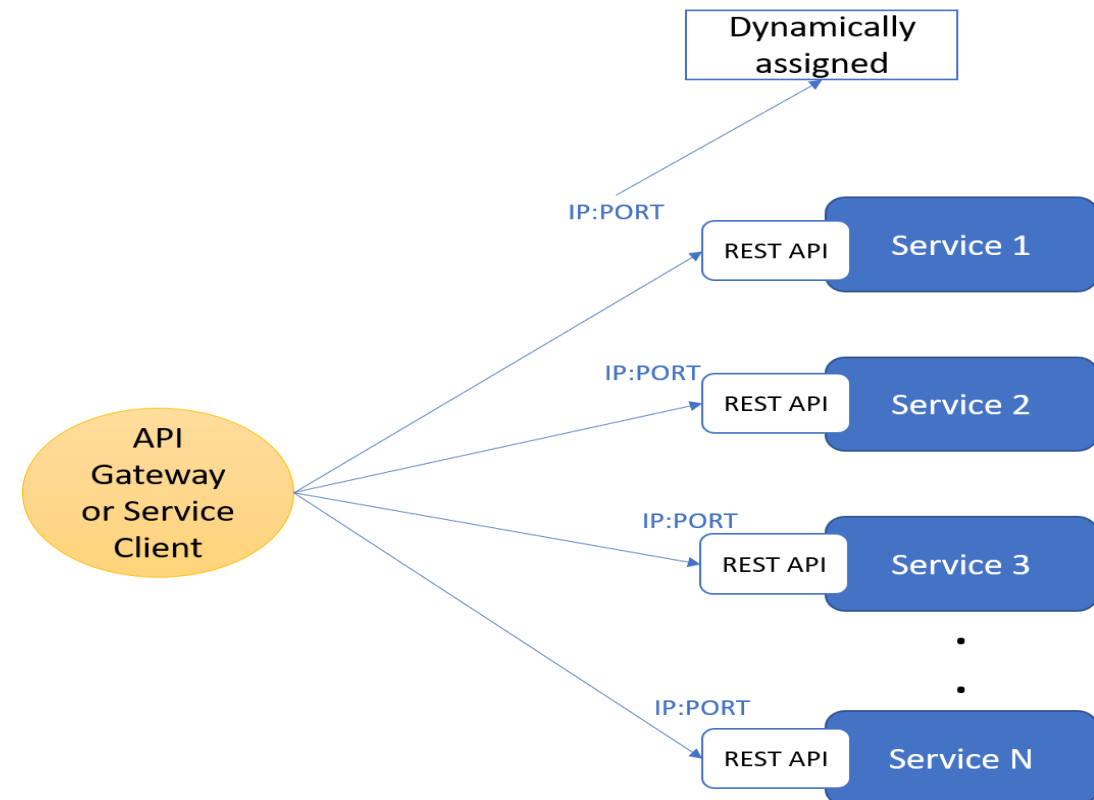
Solution -> API Gateway

- An API Gateway is a server that is the single-entry point into the system
- It encapsulates the internal system architecture and provides an API that is tailored to each client.



Introduction: What is Service Discovery ?

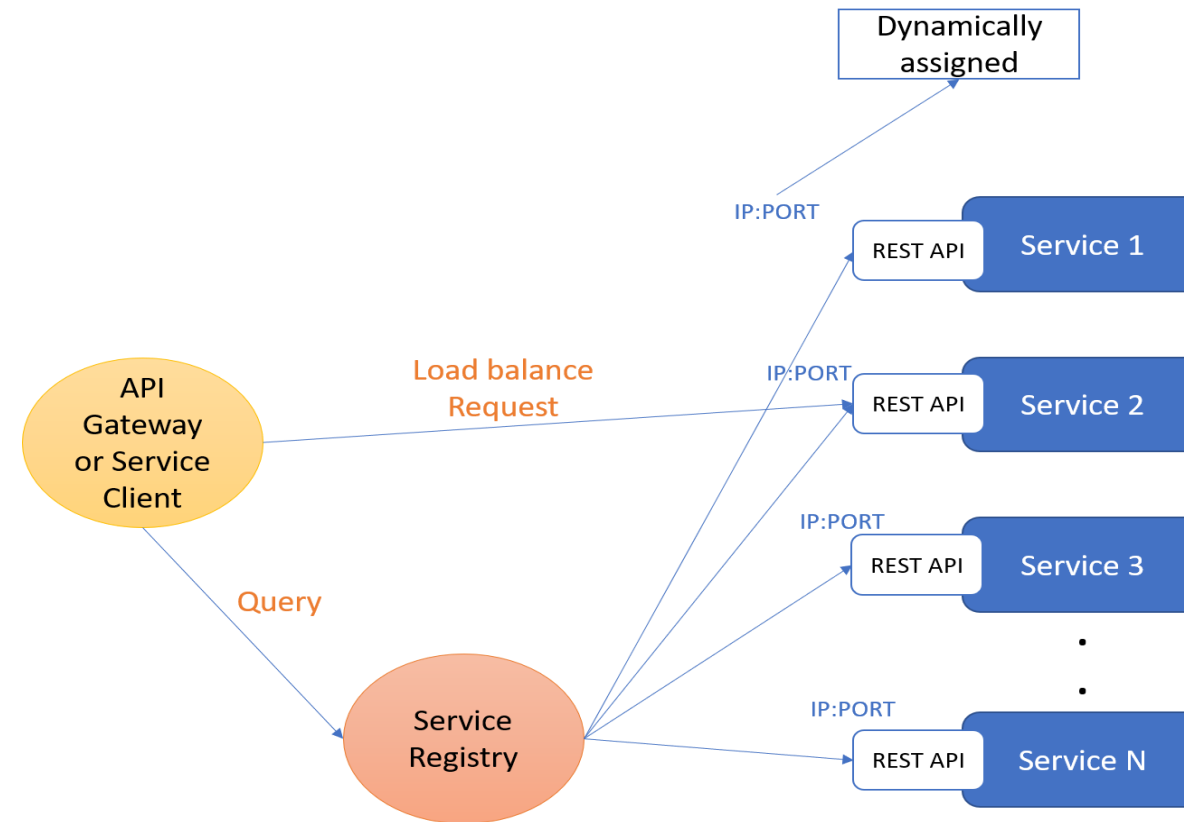
- For a microservice application doing a service request requires to know the network location (IP address and port) of a service instance.
- In a traditional application running on physical hardware, the network locations of service instances are relatively static.
- However, in a microservices application this is not the case as shown in the following diagram



To do service discovery, there are two methods

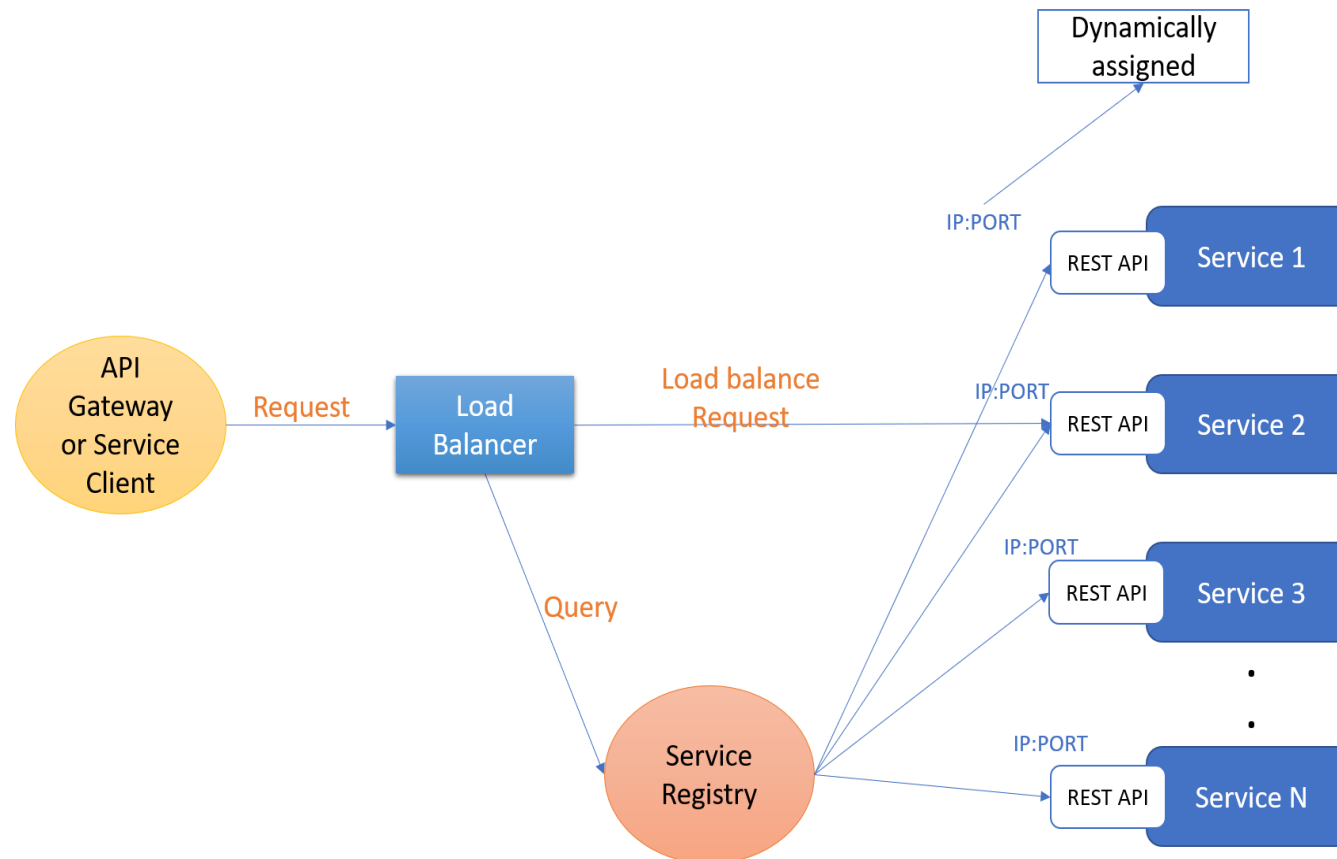
Introduction: Client-Side Service Discovery ?

When making a request to a service, the client obtains the location of a service instance by querying a Service Registry, which knows the locations of all service instances.



Introduction: Server-Side Service 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.



Introduction: Seneca.js

- A microservices framework for building scalable applications in Node.js.
- 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.

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.
- **Componentization:** Functionality is expressed as a set of plugins which can be composed together as microservices.

Introduction: Seneca.js Cont..

```
var seneca = require('seneca')()
```

Pattern

Action

```
seneca.add('role:api,cmd:hello', (msg, reply) => {  
  reply(null, {answer: "Hello "+ (msg.name)})  
})
```

Msg object

Call back function

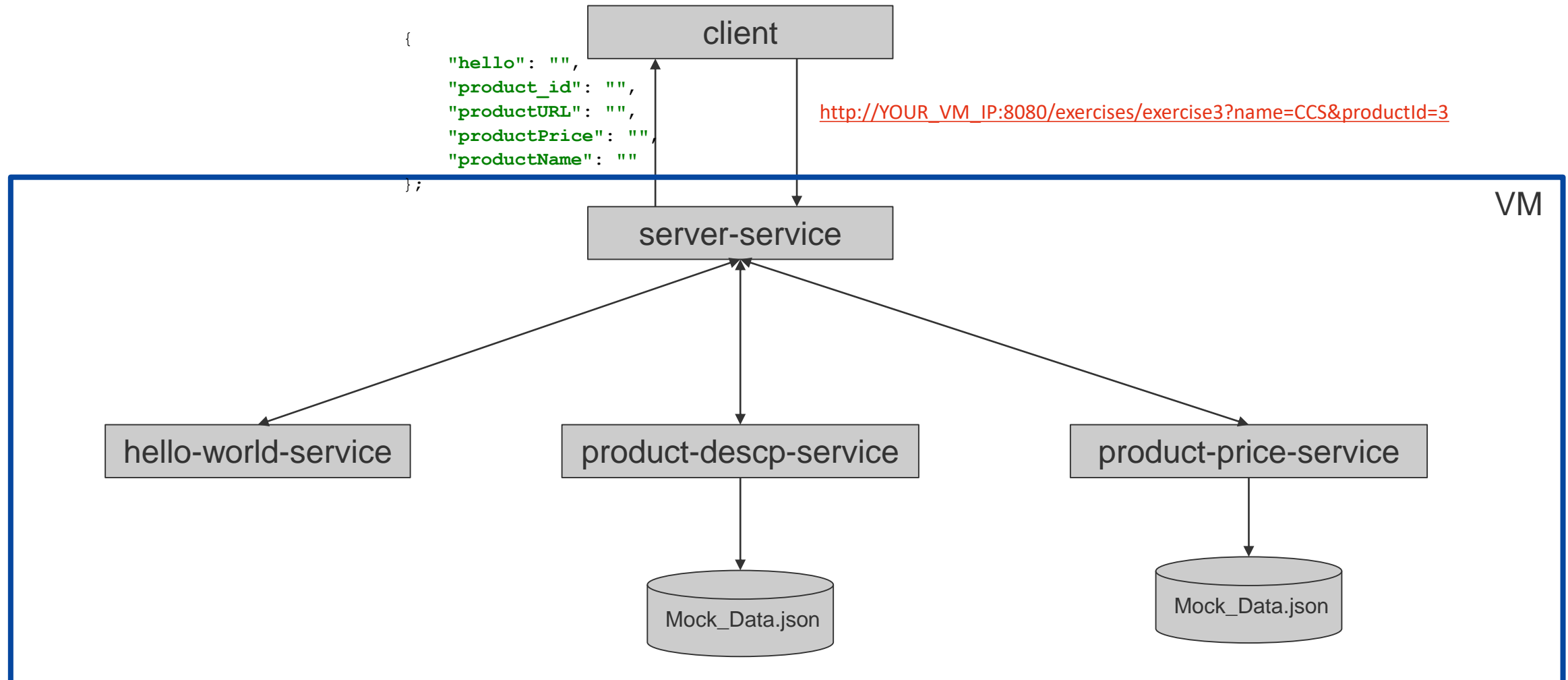
```
seneca.act({role: 'api', cmd: 'hello', name: 'CCS'},  
  if (err) return console.error(err)  
  console.log(result)  
})
```

Require for the package

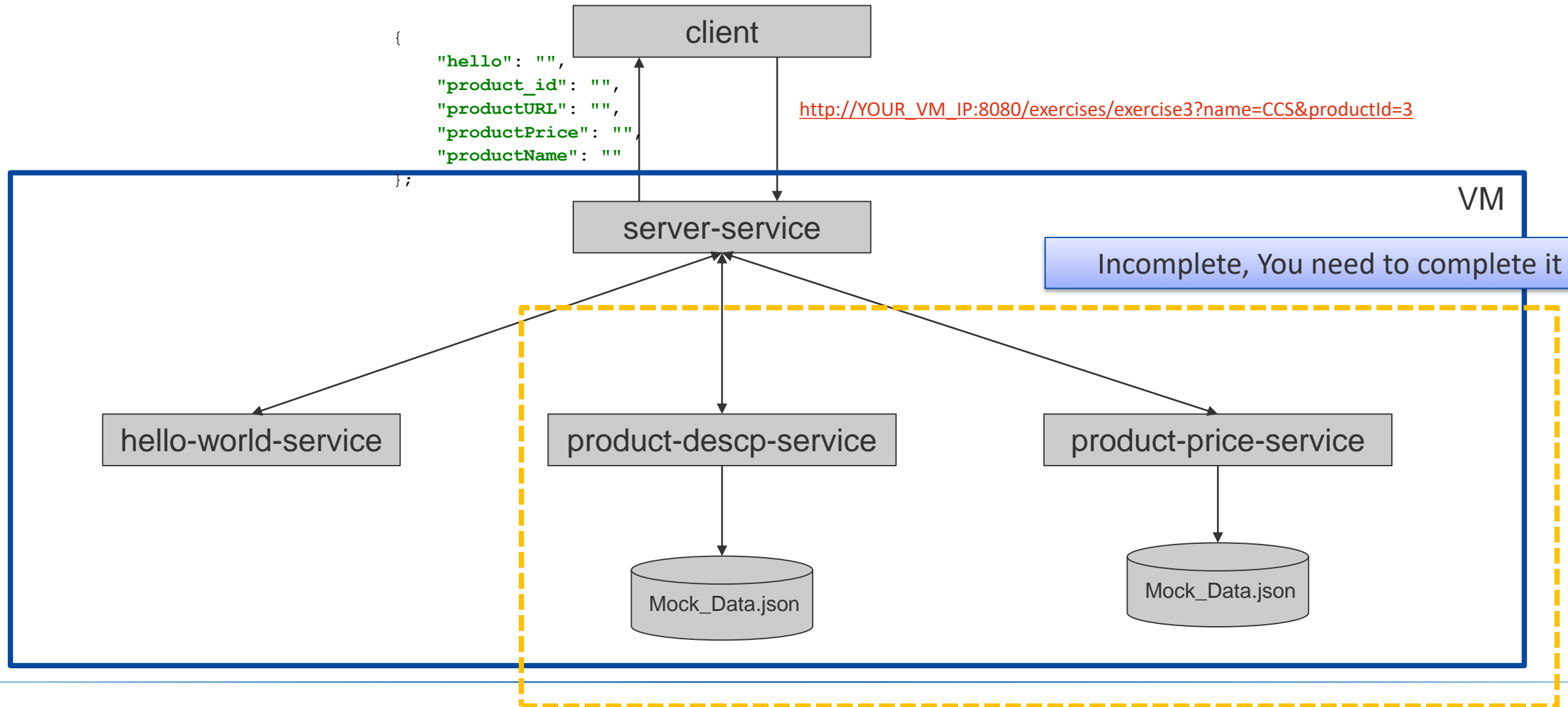
method adds a new action pattern
to the Seneca instance

submits a message to act on

Exercise 3: To Develop Microservice Architecture



Exercise 3: To Develop Microservice Architecture Cont..



Exercise 3: 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
```

To build and combine services

Hello-world-service

product-descp-service

product-price-service

server

Exercise 3: Docker Compose

Compose is a tool for defining and running multi-container Docker applications.

docker-compose.yml File

```
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
```

Exercise 3: exercises/exercise3 route

```
router.route('/exercise3')
  .get(function(req, res)
  {
    join(
      helloWorldService.sayWelcome(req.query.name),
      productDescpService.getProductURL(req.query.productId),
      productDescpService.getProductServiceName(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);
    }
  )
}
```

Exercise 3: `server/services/helloWorld.js`

```
/**
 * 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
};
```

Host and port information to send request of the service

Service Method template

Do a request to service when sayWelcome is called

Exercise 3: **hello-world-service/index.js**

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

The service running at port.

Exercise 3: `hello-world-service/helloWorld.js`

This is like a plugin and the main business logic.

```
1 module.exports = function (options) {  
2     //Add the patterns and their corresponding functions  
3     this.add('role:helloWorld,cmd:Welcome', sayWelcome);  
4  
5     //Describe the logic inside the function  
6     function sayWelcome(msg, respond) {  
7         if(msg.name){  
8             var res = "Welcome "+msg.name;  
9             respond(null, { result: res });  
10        }  
11        else {  
12            respond(null, { result: ''});  
13        }  
14    }  
15 }
```

Add the pattern and function to be called if matched

Define the function and do a callback with the result

Exercise 3: To Run the Application

1. To run the application, you need to first install **docker** and **docker-compose**. Please check previous exercise to know how to install docker.
2. 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`
3. Once docker-compose is installed, go into the root directory of the application and run the following command:
`docker-compose up`
4. If you need to build again, run this command:
`docker-compose up --build`

Exercise 3: To Run the Application Cont..

5. Run the API endpoint on the browser, the API endpoint 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": ""  
};
```

Short Demo

Tasks To be Completed

Tasks to be completed

As part of the exercise3, following are the tasks to be completed:

1. Complete the microservices **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 VM using docker-compose as explained in previous sections.
4. Check all the services running using **docker ps** command, with 4 different containers running,
5. Enable docker remote API(As done in previous exercise)

Deadline for submission: 18th December 2017 23:59

Submission

Submission

To submit your application results you need to follow this :

1. Open the cloud Class server url
2. Login with your provided 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
 1. VM ip on which your application is running

Example:

10.0.23.1

5. Then click submit.
6. You will get the correct submission from server if everything is done correctly(multiple productids will be tested while submission of the code).

Remember no cheating and no Hacking 😊

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 tasks 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.

Good Luck and Happy Coding😊

Thank you for your attention! 😊

Questions?

Appendix

Install the node and npm

- Get yourself a more recent version of Node.js by adding a PPA (personal package archive) maintained by NodeSource.

```
curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
```

- You can now install the Node.js package.

```
sudo apt-get install nodejs
```

The nodejs package contains the nodejs binary as well as npm, so you don't need to install npm separately.

- For some npm packages to work (such as those that require building from source), you will need to install the build-essentials package

```
sudo apt-get install build-essential
```

- Test Node: `node -v` (This should print a version number, so you'll see something like this v0.10.35)
- Test NPM: `npm -v` (This should print NPM's version number so you'll see something like this 1.4.28)

Installation of required modules

1. As part of this application some modules need to be installed. For installing them run the following command from inside the directory of application.

`npm install`

This command will install all the dependent modules mentioned in the package.json file.

2. If you need some other modules you can install them by running the command

`npm install "module name" --save`

This will automatically add that module in the package.json file and now you can use it inside your development file.

Node.js Client Application Deployment

1. Now your application is ready to be deployed on VM. Run the following command to start the application:

node clientApplication.js

After running this, on console you will see

“Server started and listening on port 8080”

2. Now your application is deployed on the server. Open your browser on your local machine and enter the address as

http://IP_ADDRESS_VM:8080/exercises

`'Welcome to Cloud Computing Exercises API`

Node.js Client Application Deployment : Port unblock

- If your request timed out, your VM probably has some firewall rules in place prevent a user to call your web server from the outside.
- The iptables rules are located in the file **/etc/iptables/rules.v4**. Open this file with your favourite editor:
- After line 9 insert a new line allowing incoming connections on port 8080:

`-A INPUT -p tcp -m tcp --dport 8080 -m state --state NEW -j ACCEPT`

```
# Generated by iptables-save v1.6.0 on Fri May  6 15:32:09 2016
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -i lo -j ACCEPT
-A INPUT -p icmp -m icmp --icmp-type 4 -j ACCEPT
-A INPUT -p icmp -m icmp --icmp-type 8 -j ACCEPT
-A INPUT -p tcp -m tcp --dport 22 -m state --state NEW -j ACCEPT
-A INPUT -p tcp -m tcp --dport 8080 -m state --state NEW -j ACCEPT
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -m limit --limit 3/min -j LOG
COMMIT
# Completed on Fri May  6 15:32:09 2016
~
```

- Save it and to apply the new iptables rules, you need to reload them to your local firewall system.

`iptables-restore < /etc/iptables/rules.v4`