

Cloud Computing Exercise – 3

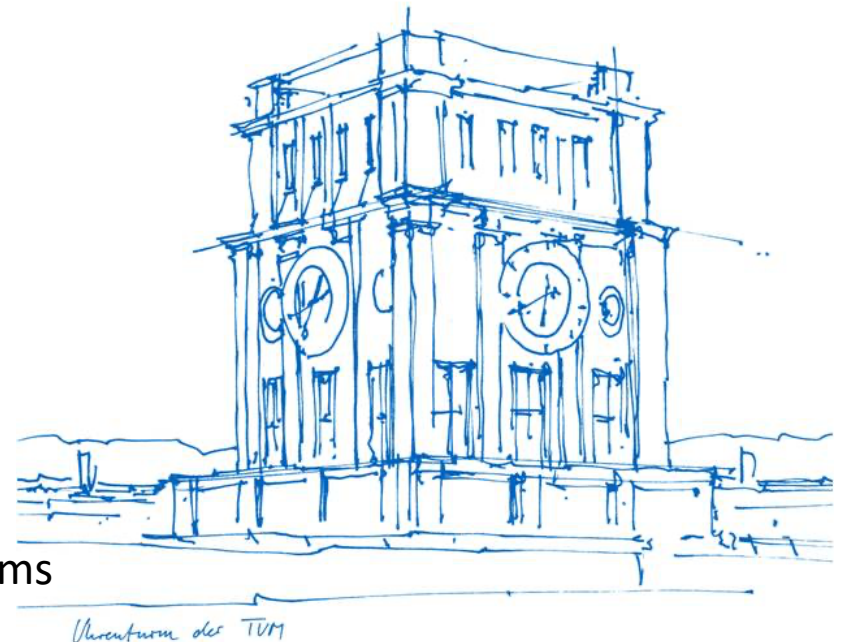
Building Microservices Application

Anshul Jindal (M.Sc. Informatics)

anshul.jindal@tum.de

Chair of Computer Architecture and Parallel Systems

Technical University of Munich (TUM), Germany



Introduction to Seneca.js

A microservices application building framework


- 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:** works based on pattern matching.
- **Transport independence:** 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.

Seneca Actions

- An **action** is a function that is identified by a JSON object.
- Actions lie at the core of Seneca.
- Actions are created using the `seneca.add` method:



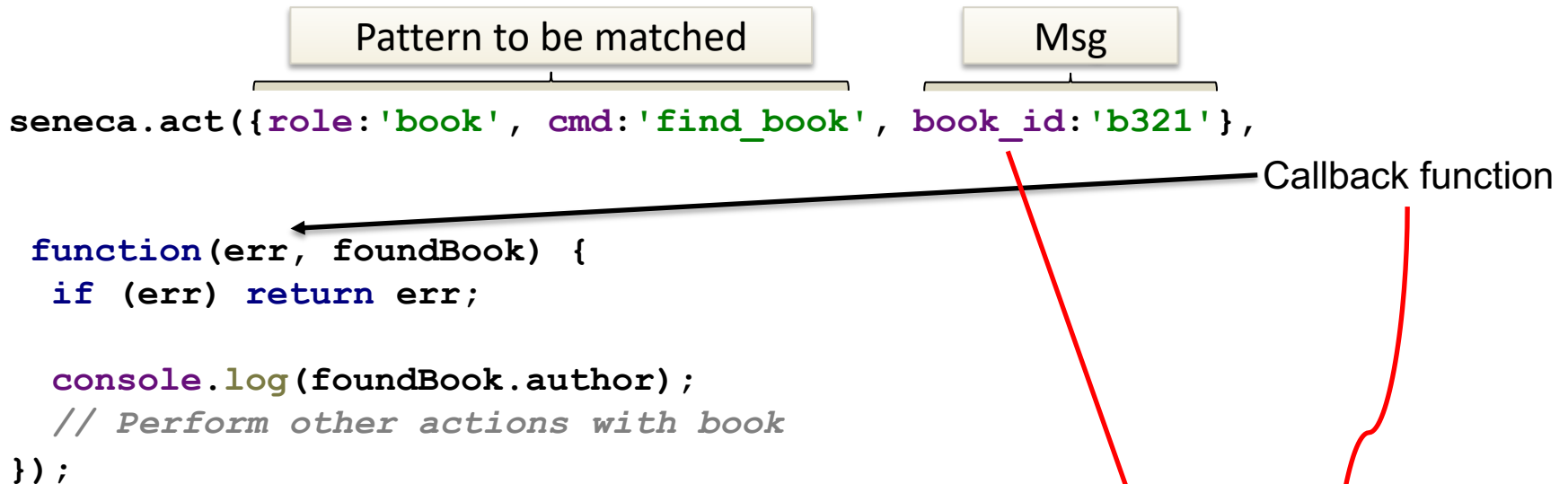
The diagram illustrates the structure of a Seneca action definition. It shows two boxes, 'Pattern' and 'Action', connected by a horizontal line. Below the 'Pattern' box, a bracket groups the first two arguments of the `seneca.add` call: `{role: 'book', cmd: 'find_book'}`. Below the 'Action' box, a bracket groups the third argument: `function(args, done) { ... }`.

```
seneca.add({role: 'book', cmd: 'find_book'}, function(args, done) {  
  var bookId = args.book_id;  
  
  // find book  
  var book = findBookInDatabase(bookId);  
  
  done(null, book);  
});
```

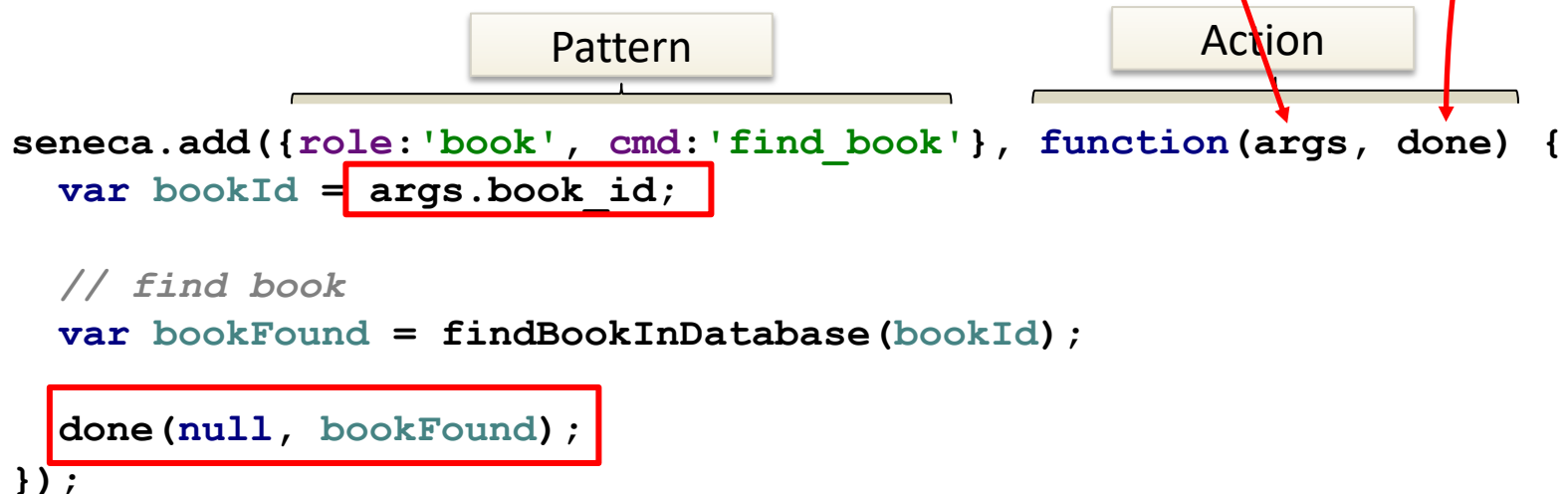
- Actions can have any granularity and any JSON pattern.
- Keep actions in the format `{role: 'namespace', cmd: 'action'}`, where “namespace” is the logical grouping of a few actions and “action” is the name of the specific action that you want to define

Seneca Actions

Calling actions can be done using the `seneca.act` method:



```
seneca.act({role: 'book', cmd: 'find_book', book_id: 'b321'},  
  
function(err, foundBook) {  
  if (err) return err;  
  
  console.log(foundBook.author);  
  // Perform other actions with book  
});
```



```
seneca.add({role: 'book', cmd: 'find_book'}, function(args, done) {  
  var bookId = args.book_id;  
  
  // find book  
  var bookFound = findBookInDatabase(bookId);  
  
  done(null, bookFound);  
});
```

Organizing Actions into files

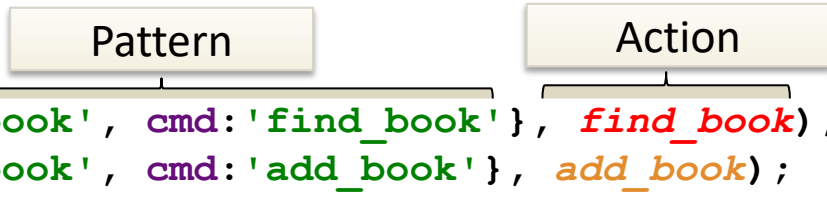
If I had a few actions that all had to do with books, I would create an **books.js** file and define all of the actions there

```
/* books.js */
module.exports = function(options) {
  var seneca = this;

  seneca.add({role: 'book', cmd: 'find_book'}, find_book);
  seneca.add({role: 'book', cmd: 'add_book'}, add_book);
  //... other action definitions

  function find_book(args, done) {
    var bookId = args.book_id;
    // ... perform find book
    done(null, foundBook);
  }

  function add_book(args, done) {
    var bookInfo = args.bookInfo;
    // ... perform book addition
    done(null, addedBook);
  }
};
```



The diagram illustrates the mapping of code to concepts. A box labeled "Pattern" points to the object literals in the `seneca.add` calls: `{role: 'book', cmd: 'find_book'}` and `{role: 'book', cmd: 'add_book'}`. A box labeled "Action" points to the function names `find_book` and `add_book`.

Organizing Actions into files

To use the **books.js** file in **server.js** file:

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

```
seneca.use('./books.js');
```

Imported module

```
var bookInfo = {  
  title: "abc", // title of the book  
  author: "ada", // name of the first author  
  //...  
};
```

Pattern to be matched

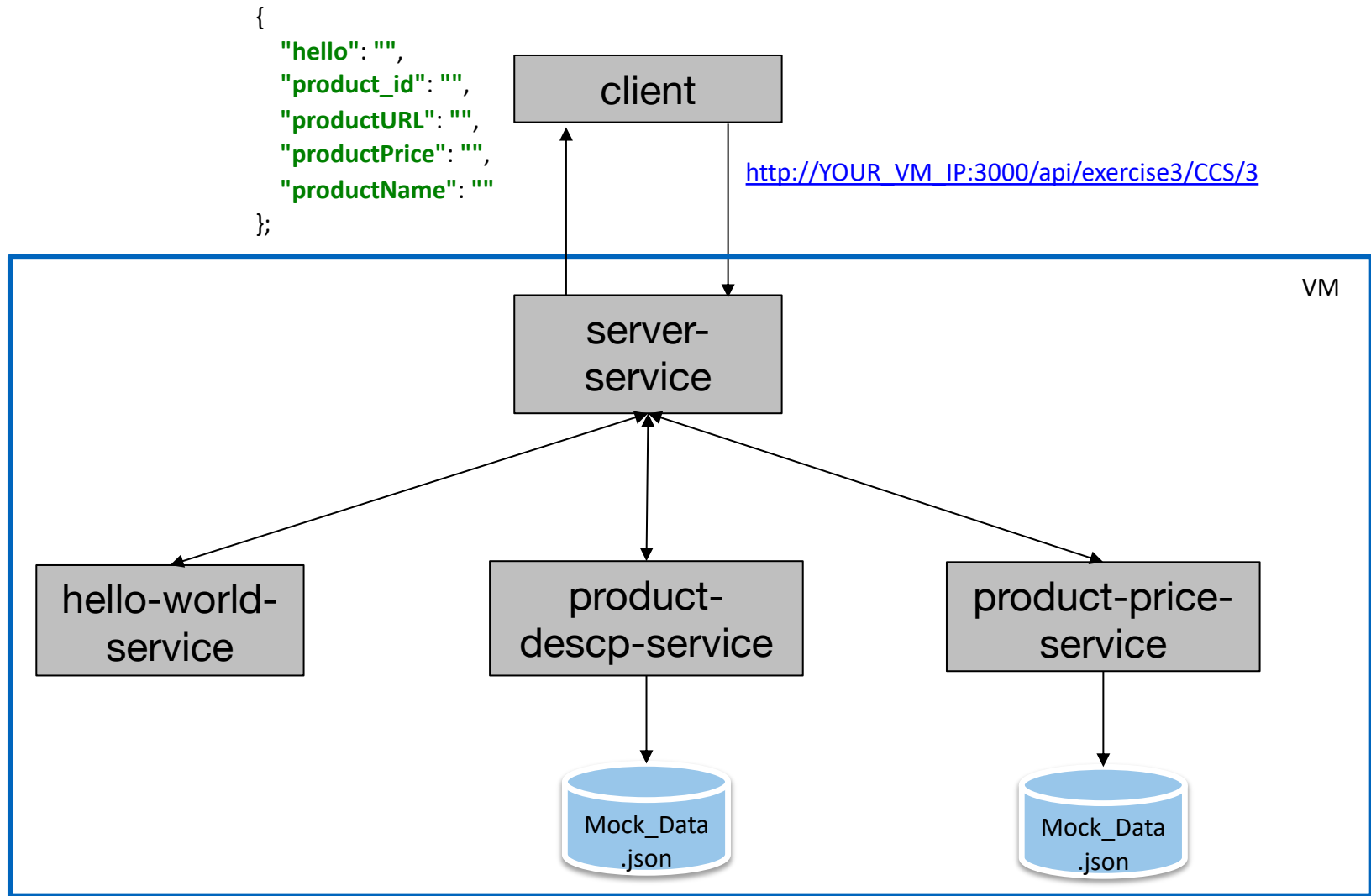
Msg

```
seneca.act({role: 'book', cmd: 'add_book', bookInfo: bookInfo},
```

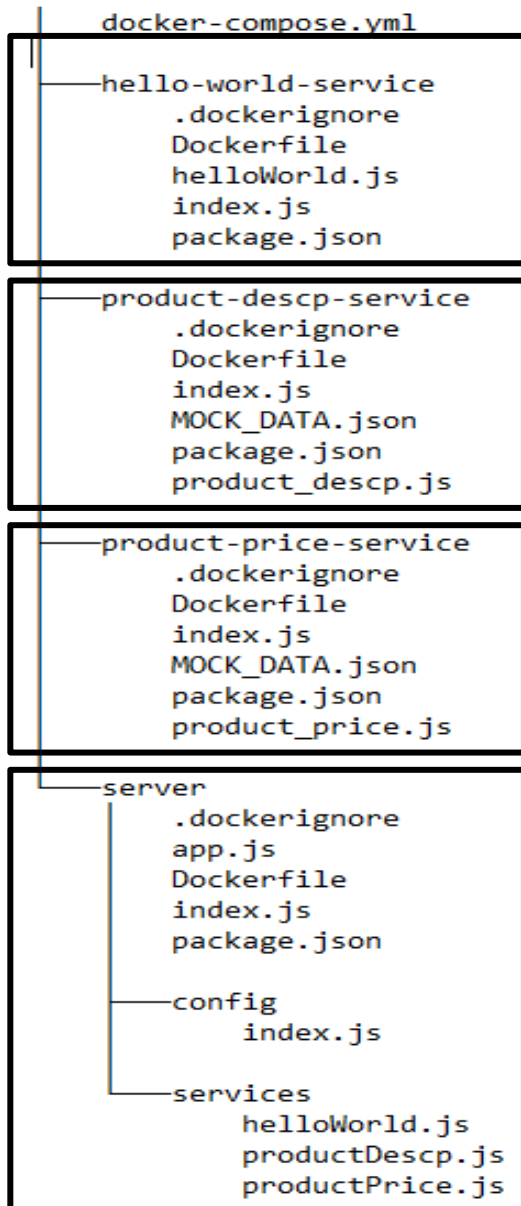
Callback function

```
function(err, book) {  
  console.log(JSON.stringify(book));  
});  
//...
```

To develop Microservice Architecture



Directory Structure of the code provided



To build and combine services

hello-world-service

product-descp-service

product-price-service

server

MOCK_DATA.json

- Contains mock information about the products.
- Array of the multiple mock products

```
[
  {
    "product_id": 1,
    "product_name": "Daltfresh",
    "product_url": "https://theglobeandmail.com/in/quis",
    "product_company": "Tagtune",
    "product_price": 42,
    "product_count": 10,
    "product_number_of_users_liked": 636
  },
  {
    "product_id": 2,
    "product_name": "Overhold",
    "product_url": "https://goo.ne.jp/commodo/placerat/",
    "product_company": "Agimba",
    "product_price": 90,
    "product_count": 92,
    "product_number_of_users_liked": 636
  },
]
```

Docker-compose.yml file

```
version: '3'
```

```
services:
```

```
  server:
```

```
    build: ./server
```

```
    image: HUB_ID/microservice:server
```

```
    ports:
```

```
      - "3000:3000"
```

```
    depends_on:
```

```
      - hello-world-service
```

```
      - product-descp-service
```

```
      - product-price-service
```

```
  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
```

Version of docker-compose file

Start of all services

server service

hello-world-service service

product-descp-service service

product-price-service service

config.js

- Configurations of all the services for the server service.
- What is the host and on what port they are running.
- Acts as our service discovery here.

```
module.exports = {  
  'server': {  
    'port': '3000',  
    'host': 'localhost'  
  },  
  'helloWorld_service': {  
    'port': '9001',  
    'host': 'hello-world-service'  
  },  
  'product_descp_service': {  
    'port': '9002',  
    'host': 'product-descp-service'  
  },  
  'product_price_service': {  
    'port': '9003',  
    'host': 'product-price-service'  
  }  
}
```

Same name as in
docker-compose.yml file

```
};
```

hello-world-service/helloWorld.js

This is like a plugin and contains the main business logic.

```
module.exports = function (options) {
```

Pattern

Action

```
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: '' });  
  }  
}
```

Function invoked when
pattern matched

hello-world-service/index.js



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

- Uses helloWorld.js
- Runs the helloWorld service at port 9001

```
module.exports = function (options) {  
    // Import the mock data json file
```

```
    const mockData = require('./MOCK_DATA.json');
```

```
    // Add the patterns and their corresponding functions
```

Pattern

Action

```
    this.add('role:product,cmd:getProductURL', productURL);  
    this.add('role:product,cmd:getProductName', productName);
```

```
    //TODO: add the pattern functions and describe the logic  
    inside the function. (Create a loop and search for the product  
    based upon the id and return its URL and name)
```

```
}
```

```
module.exports = function (options) {  
  // Import the mock data json file  
  const mockData = require('./MOCK_DATA.json');
```

```
  //TODO: Add the patterns and their corresponding functions
```

```
  //TODO: add the pattern functions and describe the logic  
  inside the function (Create a loop and search for the product  
  based upon the id and return its price)
```

```
}
```


product-descp-service/index.js

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

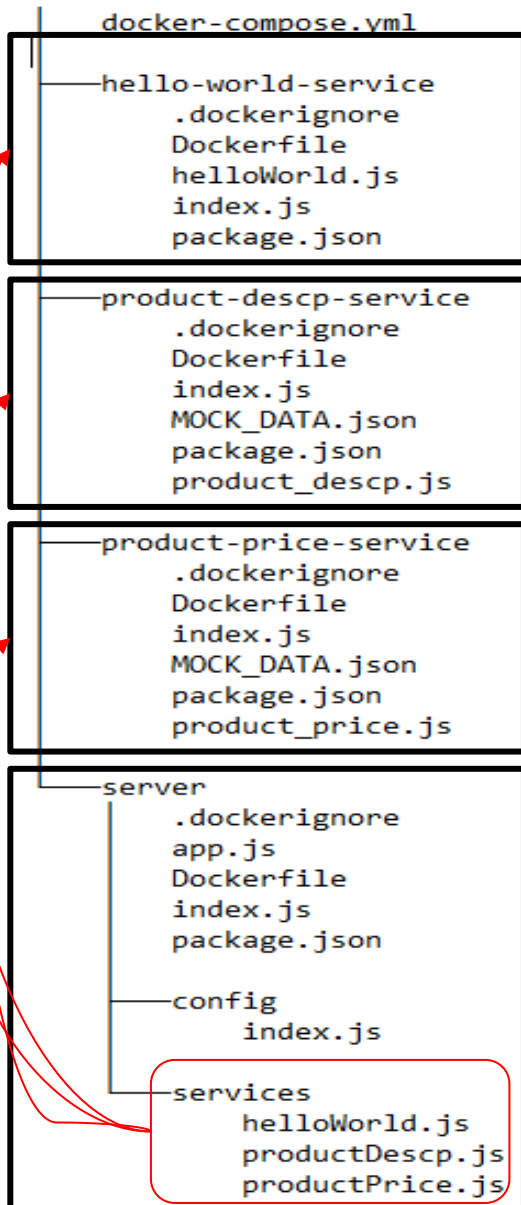
- Uses product_descp.js
- Runs the product_descp service at port 9002

product-price-service/index.js

```
require('seneca') ()  
  .use('product_price')  
  
  .listen({ port: 9003 });
```

- Uses product_price.js
- Runs the product_price service at port 9003

Microservices called from the main service



To build and combine services

hello-world-service

product-descp-service

product-price-service

server

server/services/helloWorld.js



```
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: config.helloWorld_service.port
  }).act, { context: seneca });
```

```
/**
```

```
 * Service Method
```

```
 */
```

```
const SAY_WELCOME = { role: 'helloWorld', cmd: 'Welcome' };
```

```
/**
```

```
 * Call Service Method
```

```
 */
```

```
const sayWelcome = function(name) {
  return act(Object.assign({}, SAY_WELCOME, { name }));
};
```

```
module.exports = {
  sayWelcome
};
```

Details where the service is running i.e
host and port taken from config file

Pattern

Created a function to call the
service action

name is sent to the service

Function exposed for accessible by
server/app.js

server/services/productDescp.js



```
const seneca = require('seneca')();
const Promise = require('bluebird');
const config = require('../config');
```

Details where the service is running i.e
host and port taken from config file

```
const act = Promise.promisify(seneca.client({ host:
config.product_descp_service.host, port: config.product_descp_service.port
}).act, { context: seneca });
```

```
/**
```

```
 * Service Method
```

```
 */
```

```
const GET_PRODUCT_URL = { role: 'product', cmd: 'getProductURL' };
```

```
const GET_PRODUCT_NAME = { role: 'product', cmd: 'getProductName' };
```

```
/**
```

```
 * Call Service Method
```

```
 */
```

```
const getProductURL = function(productId){
    return act(Object.assign({}, GET_PRODUCT_URL, { productId }));
};
```

```
...
```

```
module.exports = {
    getProductURL,
    getProductName
};
```

Pattern

Created a function to call the
service action

productId is sent to the service

Functions exposed for accessible by
server/app.js

server/services/productPrice.js



```
const seneca = require('seneca')();
const Promise = require('bluebird');
const config = require('../config');

/**
 * Convert act to Promise
 */

const act = Promise.promisify(seneca.client({ host:
config.product_price_service.host, port:
config.product_price_service.port })).act, { context: seneca });
```

Details where the service is running i.e
host and port taken from config file

```
/**
 * TODO: Define Service Method
 */
```

```
/**
 * TODO: Call Service Method
 */
```

```
const getProductPrice = function(productId) {
```

Created a function to call the
service action

productId is sent to the service

```
  /**
   * To DO: Write act Method
   */
```

```
};
module.exports = {
  getProductPrice
};
```

Function exposed for accessible by
server/app.js

```
const express = require('express');  
/**  
 * import the Services we need  
 */  
const helloWorldService = require('./services/helloWorld');  
const productDescpService = require('./services/productDescp');  
const productPriceService = require('./services/productPrice');  
/**  
 * javascript promises for join function  
 */  
const join = require("bluebird").join;
```

Import all required services

```
const app = express();
```

```
const router = express.Router();
```

Creating a router

```
...
```

```
app.use('/api', router);
```

All the endpoints having
<http://localhost:3000/api>
Will be sent to **router**

/exercise3/:name/:productId



```
router.route('/exercise3/:name/:productId')
  .get(function(req, res)
  {
```

```
    join(
```

```
      helloWorldService.sayWelcome(req.params.name),
      productDescpService.getProductURL(req.params.productId),
      productDescpService.getProductNames(req.params.productId),
      productPriceService.getProductPrice(req.params.productId),
      function (resultHelloWorld, productDescpServiceURL,
        productDescpServiceName, productPriceServicePrice ) {
```

```
        var ex3_response_message = {
          "hello": resultHelloWorld.result,
          "product_id": req.params.productId,
          "productURL": productDescpServiceURL.result,
          "productPrice": productPriceServicePrice.result,
          "productName": productDescpServiceName.result
        };
      }
```

```
      res.send(ex3_response_message);
    }
  );
});
```

Query all services and
get their responses

Sending query
Params

Send Result back to
user

Installation and Running the application

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

`docker-compose up`

3. If you need to build again, run this command:

`docker-compose up --build`

4. Run the API endpoint on the browser, the API endpoint format will look something like this (productid can be changed):

http://YOUR_VM_IP:3000/api/exercise3/CCS/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": ""  
};
```

name (user-defined)

Don't forget to enable port 3000!

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/productPrice.js](#)
 - c. [product-descp-service/product_descp.js](#)
 - d. [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 section. Or:
 - a. You can push the images from the local laptop using docker-compose push
 - b. Then copy the docker-compose.yml file to VM and remove all the build lines.
 - c. Then run docker-compose up
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: Check the submission server

Submission

Submission Instructions

To submit your application results you need to follow this :

1. Open the Cloud Class server url : <https://cloudcom.caps.in.tum.de/>
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
 - 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).
7. **Don't forget to enable ports 3000 and 4243**

Remember no cheating and no Hacking 😊

Important Points

1. Make sure your VM and your application is running after following all the steps.
2. You will get to see, what your application has submitted to the server and what is the response whether successful or not.
3. You can submit as many times until the deadline of exercise.
4. Multiple submission will overwrite the previous results.

If you found any bug or you have any suggestions please report to me!

Good Luck and Happy Coding😊

Thank you for your attention!
Questions?