## **Designing and Implementing Distributed, Service Based Architecture**

## **Exercise Manual**

## **Table of Contents**

[Getting Started: 1](#_Toc495421114)

[Identifying Microservices 2](#_Toc495421115)

[Identifying Service Interfaces 3](#_Toc495421116)

[Defining a Service Interface with Swagger 4](#_Toc495421117)

[Getting Started with Docker 6](#_Toc495421118)

[Creating a Docker Image 8](#_Toc495421119)

[Building a Service Aggregation 9](#_Toc495421120)

[Mongo DB 13](#_Toc495421121)

[Exploring Cassandra 15](#_Toc495421122)

[Neo4J 18](#_Toc495421123)

[Docker Compose 19](#_Toc495421124)

[Description of the Membership Application 21](#_Toc495421125)

# Getting Started:

The best way of running the course material is to clone the Git repository at [https://](https://mikejrway@bitbucket.org/mikejrway/bbc_micro_services_load.git)mikejrway@bitbucket.org/mikejrway/bbc\_micro\_services\_load.git

git clone [https://](https://mikejrway@bitbucket.org/mikejrway/bbc_micro_services_load.git)mikejrway@bitbucket.org/mikejrway/bbc\_micro\_services\_load.git

The directory into which you clone the repository will be referenced as $COURSE\_HOME in the rest of this manual.

Alternatively, your instructor can configure a virtual machine on the Amazon AWS cloud for you (access to this will be much slower than your local machine).

If you need to access the virtual machine, the credentials are:

User: ubuntu

Password: ubunTwo

# Identifying Microservices

## **Objective**

In this exercise you will derive a set of candidate services for an application based upon a simple description of the applications desired functionality

## **Overview**

The exercise is best performed in groups of 4-6 people. Using the application description at the end of this manual as a starting point, define an initial (candidate) set of services for the application.

## **Detailed Steps**

1. Examine the description of the Membership Application at the end of this document and play “spot the noun” this should yield a list of all possible services.
   * Write the list of services down as the input to the next phase.
2. Refine the list. You should consider:
   * Does each service map onto a real world concept?
   * Is it likely that creating a service will provide a benefit to the system (in terms of flexibility or scalability)?
3. For each of the user / system interactions outlined in the description, consider if the services provide the capabilities needed.
   * Add any extra services identified to our system



# Identifying Service Interfaces

## **Objective**

In this exercise, you will refine the definition for a single service to identify a service interface for that service.

## **Overview**

In our version of the solution to the previous exercise, we have identified a member service. Examining the service, we have identified these use-case titles for the service:

* Add new member
* Update member details
* List members
* Get details of member
* Archive member

## **Detailed Steps**

1. Examine the list of use-cases, decide and document:
   1. What operations the service should support
   2. What the request and response messages for those services should look like
   3. Whether the methods should use GET, POST, PUT, DELETE or PATCH (there is additional information on when to use these methods at the end of this exercise)
2. Create an output document identifying

## **Bonus: If you have more time**

Other parts of the system need to know that member details have been modified. Consider the operations you would need to support some form of publish / subscribe interface to support this notification.

## **Background Information**

## Use of GET, POST, PUT, DELETE or PATCH

(source restcookbook.com)

**GET** is used to retrieve a resource

**DELETE** is used to delete (or perhaps archive) a resource.

**PUT** is used when you can update a resource completely through a specific resource.

**POST** is used when adding or updating a resource when you do not have a location (resource identifier) for the new resource.

**PATCH** is used to perform an update on part of a resource.



# Defining a Service Interface with Swagger

## **Objective**

In this exercise, you will define an interface for a service using OpenAPI / Swagger

## **Overview**

You will use the on-line swagger editor at swagger.io to create a definition for a service.

The definition should describe this service

Service Name : Member service

Path /

Methods:  
 GET  
 Returns an array of all members

POST  
 Adds a new member

Path /{memberId}

Methods:

GET  
 Returns an object representing a member  
 PUT   
 updates a member  
 DELETE

Deletes a member  
Schema

Member

id : int64

name : string

address : string

email : string

## **Detailed Steps**

* Open swagger.io in a browser
* Select tools / editor – online editor
* Create a swagger specification for a Member service
* Creating your first OpenAPI definition is quite challenging, we suggest you use an existing definition as a template. We have provided one of the standard Swagger samples at :   
  $COURSE\_HOME/Source/swagger/heroko-swagger.yaml
* Make sure that there are no errors in the definition. If you are new to Swagger / YAML, fixing the errors can be quite a challenged! Please ask for help if needed.

## **Bonus: If you have more time: create a service from the definition**

* You service definition must be error free before you attempt these steps.
* Make sure that the host: is set to localhost:8095
* From the Swagger menu select **Generate Server | NodeJS-Server.**
* Unzip the zip file onto your file system (in the exercises folder).
* Open a terminal session and change to the nodejs-server-server folder you have just unzipped and run **npm start.**
* When the nodeJS session has finished initializing open <http://localhost:8095/docs>
* You should now be able to test the skeleton of your service.

## **Bonus: If you have even more time**

* Open the nodejs-server-server folder with Visual Studio Code.
* Locate the controllers/DefaultService.js file.
* Modify the example data for some of the methods.
* Stop the server (CTRL + C in the terminal window)
* Re-run the server
* Use the test tool <http://localhost:8095/docs> to explore the service and check you see the new data.



# Getting Started with Docker

## **Objective**

In this exercise, you will gain experience with the basic functionality of Docker and the Portainer management tool.

## **Overview**

## **Detailed Steps**

* Start a Docker machine from an image

docker run --name lamp -d -p 80:80 tutum/lamp

* Examine the web output with a browser by opening <http://localhost/>
* Stop the container

docker stop lamp

* Refresh the browser to check the container stopped. You should not see a web page.
* Restart the container

docker start lamp

* Connect to the container with a terminal (running bash).

docker exec -it lamp bash

* Execute the command ps x to see the running processes
* Type exit to leave the bash shell

## Bonus: If you have more time

* Start the Portainer management tool in a Docker container

docker run -d -p 9000:9000 -v /var/run/docker.sock:/var/run/docker.sock portainer/portainer

* Open [http://localhost:9000](http://localhost:9000/)
* When prompted specify a password of 'password' click Validate
* At the login screen use your new password
* At the next screen select **Manage the Docker instance where Portainer is running.** Then **Connect**.
* You can now use Portainer to explore Containers, Images and Volumes. Feel free to experiment. If you kill off the Portainer container by accident, just use the above commands to restart it.



# Creating a Docker Image

## **Objective**

In this exercise you will create a new docker image from a base container image.

## **Overview**

You will create an initial container using the library/httpd image. You will then use this as the starting point for creating a custom image by defining a dockerfile

## **Detailed Steps**

1. Examine the documentation for the base image at <https://hub.docker.com/_/httpd/>
2. Does the documentation explain how to configure a basic static HTML site?
3. Create an empty directory. In the directory, create an empty file called Dockerfile and an empty directory called public-html
4. In the public-html directory, create an index.html file with minimal content
5. Using the dockerfile reference ([https://docs.docker.com/engine/reference/builder/](https://docs.docker.com/engine/reference/builder/#from)) as a a guide
   1. Add a FROM statement to the start of the dockerfile specifying that you want to use version 2.4 of the httpd image
   2. Add a statement to copy the public-html directory to /usr/local/apache2/htdocs/ on the image
6. Open a terminal to the directory holding your dockerfile
7. Build the image with the command:   
   docker build -t httpd-exercise .
8. Run the image using   
   docker run --name httpdemo -p 80:80 --rm httpd-exercise
9. Open a browser and go to localhost
10. You should see your HTML page displayed

## **BONUS: If you have more time**

1. Open a new terminal and run the command

docker exec -it httpdemo bash

1. This opens a bash shell onto the container
2. Use the shell to explore the container file system
3. Try and edit your index.html page with the nano editor (it’s in htdocs).
   1. Hint: nano index.html
   * That did not work as nano is not installed!
4. Stop the httpdemo container. You can do this through Portainer or with the command docker stop httpdemo
5. Modify the dockerfile to install nano by adding the RUN commands
   * RUN apt-get update
   * RUN apt-get install -y nano
6. Rebuild the container image
7. Run the image again
8. Connect to the image with a bash shell and try to edit the index.html file. Change the message and then check the changes appear in your browser.



# Building a Service Aggregation

## **Objective**

To create a service which aggregates the result from invoking other services.

## **Overview**

We have created a docker image with 3 microservices on it. These are skeleton services configured specifically for this exercise.

The services are:

Member Service: http://localhost:8090/docs

Subscription Service: http://localhost:8091/docs

Purchase History Service: http://localhost:8092/docs

In the exercise, you will create a service that aggregates these 3 services.

The new service is the Member-Subscription-Purchase history service. The operation of this service is:

* For the supplied memberId (input with the request) retrieve the details of the member from the Member service (we just want their name).
* Get a list of purchases for the memberId from the Purchase History service.
* For each purchase, get the description of the subscription from the Subscription service.
* Return an output as shown:
  + {   
     “memberId”: 0,  
     “memberName”: “abcd”,  
     “email”: “x@y.com”

“purchases”: [  
 “subscriptionId”:0,  
 “subscriptionName”: “abcd”,  
 “price”: 0.0  
 ]  
}

## **Major Steps you will be asked to perform**

* **Generate a service stub**
* **Start the docker container with the services to aggregate**
* **Generate clients for the 3 services**
* **Integrate the clients into the service stub to create the aggregate service**

## **Detailed Steps**

**Generate a service stub**

* You now have a choice: you can perform the remainder of the exercise with JavaScript / NodeJS which is what our instructions are based upon or you can be more ambitious and create the service with your preferred platform (though you need to check that that Swagger supports it).
* Load the swagger.yaml file from bbc\_micro\_services\_load\Exercises\ServiceAggregation into the Swagger editor.
* Using the Swagger editor, create a **nodejs-server** implementation of the service.
* Unzip the generated code onto your disk (either the cloud based VM or your local machine if you have the correct environment).
* If you are using our environment, then open the directory you have just created with Visual Studio Code.
* If you are using the NodeJS service then you can quickly test the code by running **npm start** in the directory you created.

**Start the docker container with the services to aggregate**

* Start the services you will aggregate with:
  + **docker run -p 8092:8092 -p 8091:8091 -p 8090:8090 bbcdemo/services-to-aggregate**
* Verify that the services have started by visiting http://localhost:8090/docs
* The JSON version of the OpenAPI specs for each service are available at http://localhost:XXXX/api-docs for each service.

**Generate clients for the 3 services**

* You will need 3 directories for the clients within your project
  + If using the nodejs server code then create them under the root of your project as:
    - member-client
    - purchases-client
    - subscription-client
* Start with the Member Service client
  + Locate the swagger.json file at <http://localhost:8090/api-docs>
  + Copy the swagger.json into the Swagger editor (you can use Edit | Convert to YAML if you wish)
  + Generate a client for the service
    - Select javascript as the type if following our instructions
  + From the client zip file extract everything from the src folder into the member-client folder
* Repeat the above steps for the purchases-client (port 8092) and the subscription client (port 8091)

**Integrate the clients into the service stub to create the aggregate service**

* Depending on the client you have built there should be documentation on how to incorporate the client into your code most likely in a README.md file within the downloaded zip file.
* If you are building the NodeJS service then use these steps:
  + Open controllers/DefaultService.js
  + Make the 3 service clients available with:

let MemberService = require('../clients/member-client/index');

let SubscriptionService = require('../clients/subscription-client/index');

let PurchaseService = require('../clients/purchases-client/index');

* Create 3 functions to access the services

function getMemberData(memberId)

function getSubscriptionData(memberId)

function getPurchaseData(memberId)

* Each of these methods should invoke it’s respective client and return a Promise holding the returned data

Here is an example of the getMemberData() function as a template

function getMemberData(memberId) {

let promMember = new Promise((resolve, reject) => {

let api = new MemberService.DefaultApi();

api.memberIdGET(memberId, (error, data, response) => {

if (error) {

console.error(error);

reject(error);

} else {

console.log('API called successfully. Returned data: ' + data);

resolve(data);

}

});

});

return promMember;

}

* Our instructions are intended as an outline: if you need more help, there is a solution at $COURSE\_HOME/Exercises/Solutions
* Use Promise.all() to wait for the completion of all of the service calls.
* Populate a results object with the values returned from calling all 3 services.
  + The NodeJS code includes an example of how the data should look (and don’t forget you can use the solution as a guide).
* Test you work by running npm start then visiting the displayed URL
* Use the Swagger UI tool to create a request with a memberId of 1



# Mongo DB

## **Objective**

To investigate the capabilities of the MongoDB NoSQL database.

## **Overview**

You will run MongoDB in a Docker container and then explore some of its capabilities.

## **Detailed Steps**

* Start the Docker container with the command

docker run --name mongo-container -v $COURSE\_HOME/Resources:/mydata -p 27017:27017 -d mongo

* Connect a terminal to the container so you can enter commands

docker exec -it mongo-container bash

* In the terminal session type the following open a mongo session and to specify a database to use:

mongo

use testdb

* Create some variables using OLN which you will then insert into the database

myplace = { name :"myplace", location: { x: 3, y: 5 } }

yourplace = { name : "yourplace", location: { x: 1, y: 2, z: 4 } }

other = { name : "other", location: { x: -1, z: 4 } }

* Insert the data values into the database in a collection called locations

db.locations.insert( myplace )

db.locations.insert( yourplace )

db.locations.insert( other )

* Perform a query to retrieve all locations

db.locations.find()

* Perform a query to locate myplace

db.locations.find({name:"myplace"})

* Perform a query to locate all locations where the x value is 1

db.locations.find({"location.x" : 1 })

* Create a query for locations with a name of "other" or a z value of 4 (there is a command reference at <http://docs.mongodb.org/manual/tutorial/query-documents/>)
* Exit the mongo terminal with Ctrl+C

## Bonus: Explore some real data

* Perform a bulk import of some real data

mongoimport --db dbsubs --collection subs --type csv --headerline --file /mydata/subscriptions.csv

* Explore the data with the following commands

mongo  
use dbsubs  
db.subs.find();

* Create and execute queries to find:
  + Members whose names begin with Michael
  + Members whose Paid\_InclTax value is less than 40

## Additional Bonus:

* Integrate MongoDB with your own code



# Exploring Cassandra

## **Objective**

To investigate the capabilities of the Cassandra NoSQL database.

## **Overview**

You will run Cassandra in a Docker container and then explore some of it’s capabilities.

## **Detailed Steps**

* If it is still running, stop the mongo container with

docker stop mongo-container

* Start the Docker container with the command

docker run --name cassandra -v $COURSE\_HOME/Resources:/mydata -d -p 9042:9042 cassandra

* Connect a terminal to the container so you can enter commands

docker exec -it cassandra bash

* Start the cassandra query shell

cqlsh

(May have to wait for this to work and then retry)

* Create a Keyspace (database)

CREATE KEYSPACE subscriptions WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy', 'datacenter1' : 2 };

* Make the new Keyspace current

use subscriptions;

* Create a table to hold our subscription data

CREATE TABLE subscriptions ( id varchar, name varchar, member\_id varchar, type varchar, renew boolean, startdate varchar, enddate varchar, purchasedate varchar, paid decimal, PRIMARY KEY(id));

* Import the subscription data from a CSV file

COPY subscriptions (id , name , member\_id , type , renew , startdate , enddate, purchasedate, paid ) FROM '/mydata/subscriptions.csv';

* Query the imported data

select \* from subscriptions;

* Attempt to filter the data

select name from subscriptions where type='Out Of Port';

* Create an index to enable the filter

create index on subscriptions(type);

* Repeat the previous query – it should now work

### Bonus: Programmatic Interaction with Cassandra

We have created a NodeJS client for you at $COURSE\_HOME/Exercises/Cassandra

* Open the folder with VSCode.
* Examine index.js – we have created a very simple client to the Cassandra database.
* The code is currently just performing the same queries we used in the previous part of the exercise.
* Run the code by selecting Debug | Start Debugging from the VSCode menu.
* You should see the lists of names output
* If you have time: modify the code to further filter the members by those who have paid less than or equal to 40.00
  + (You will need to add an index).



# Neo4J

## **Objective**

To investigate the capabilities of the Neo4J NoSQL database.

## **Overview**

You will run Neo4J in a Docker container and then explore some of its capabilities. In this case, due to the complexity of getting started with Neo4J we will use their tutorial examples as a guide to its operation.

## **Detailed Steps**

* If it is still running, stop the cassandra container with:

docker stop cassandra

* Start the Neo4J container with the command.

docker run -p 7474:7474 -p 7687:7687 --volume=$HOME/neo4j/data:/data neo4j

* Open [http://localhost:7474](http://localhost:7474/) (or connect to the virtual machine from your localhost by specifying the host name in AWS).
* Use an initial password of neo4j
* On the password change screen use password as the new password!
* Neo4J is supplied with a nice getting started tutorial. It would be pointless our creating one!
* The following steps guide you through the operation of the tutorial.
* The opening screen shows a command line (at the top) and 3 boxes.
* Click on the **Write Code** button.
  + A new box should open giving you the choice of creating a Movie Graph or working with the Northwind database.
  + Select the Movie Graph
* Click **Create a Graph**.
  + You can close any of the boxes on the screen which have an **X** in the top right corner. Keep the Movie Graph box open through the following steps.
* Read the text and when ready, click the **>** button.
* Again, examine the text and look at the query. Click in the query text box to transfer it to the command window.
* Click the Run button to the right of the command window to execute the query.
* Examine the result of the query (a node graph) and when ready, scroll down to the The Movie Graph box and click the > arrow to move to the next step.
* This presents a series of queries for you to try following a similar procedure to above.
* Work your way through the tutorial.



# Docker Compose

## **Objective**

In this exercise you will use Docker Compose to create a set of containers which host the 3 services we aggregated earlier.

## **Overview**

In the earlier exercise, the 3 services were running in a single container for simplicity. This is not good practice. In this exercise, you will create 3 separate containers and then create an orchestration using Docker Compose.

## **Detailed Steps**

**Create the Docker images to Orchestrate**

* Create 3 folders under Exercises/Composition called MemberService, SubscriptionService and PurchaseService.
* In each of these create a file called Dockerfile with the content:

FROM node:6-onbuild

EXPOSE XXXX

This creates a new Docker image using the standard Docke NodeJS image <https://hub.docker.com/_/node/>. It is particularly elegant as it requires no other configuration to run our NodeJS service applications.

* Copy the contents of $COURSE\_HOME/Source/docker/ServicesToAggregate/SimpleMembers to your MemberService directory.
* Open the index.js file and identify the serverPort (line 8). Replace the XXXX value in the Dockerfile with the correct port number.
* Create a Docker Image with the command:

docker build -t member-service .

* Copy the contents of $COURSE\_HOME/Source/docker/ServicesToAggregate/SimpleSubscription to your SubscriptionService directory.
* Open the index.js file and identify the serverPort (line 8). Replace the XXXX value in the Dockerfile with the correct port number.
* Create a Docker Image with the command:

docker build -t subscription-service .

* Copy the contents of $COURSE\_HOME/Source/docker/ServicesToAggregate/SimplePurchaseHistory to your PurchaseService directory.
* Open the index.js file and identify the serverPort (line 8). Replace the XXXX value in the Dockerfile with the correct port number.
* Create a Docker Image with the command:

docker build -t purchase-service .

**Create the composition**

In Exercises/Composition open the template docker-compose.yaml file we have provided.

* Modify it to define 3 services:
  + member
  + subscription
  + purchase
* Each of these services should specify the correct image (that you have just created) and expose the port matching the one specified in their Dockerfile.
* To avoid using excess resources, set the replicas value to 1 (or delete the property).

**Test your work**

* Run the command docker-compose up in the Exercises/Composition folder.
* Visit <http://localhost:8090/docs/>, <http://localhost:8091/docs/>, <http://localhost:8092/docs/> to validate that your services are running correctly.

## 

# Description of the Membership Application

This is a very simplistic description of the application requirements. Please feel free to make assumptions and / or ask your instructor for clarification.

The application manages member details and the products to which they subscribe:

**Provides the following User Interfaces (UIs)**

1) Self-registration / detail editing for members

2) End user facility to add / delete subscriptions

3) Admin to allow the management of subscription products

4) Payment UI

5) Purchase history

**Provides API’s**

1) Query if a particular member has a subscription

2) List all members holding a subscription

3) List all members

4) List details for a member

Has a time or event driven component which notifies users if subscriptions are due to expire