## **Objective**

## **Overview**

## **Detailed Steps**

# 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 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

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.

## 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.

# Docker Ex 1

Start a Docker machine from an image

Connect to it.

Start stop / explore

Then use Portainer:dmjrw

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

# Docker Ex 2 – Slide 101

## Objective

In this exercise you will create a new docker image from a base container image. You will use the library/library httpd image.

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/" \l "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

1. Open a browser and go to localhost
2. You should see your HTML page

BONUS

1. Open a new terminal and run the command docker exec -it httpdemo bash
2. This opens a bash shell onto the container
3. Use the shell to explore the container file system
4. 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!
5. Stop the httpdemo container. You can do this through Portainer or with the command docker stop httpdemo
6. Modify the dockerfile to install nano by adding the RUN commands
   * RUN apt-get update
   * RUN apt-get install -y nano
7. Rebuild the container image
8. Run the image again
9. 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.

======================================================================

/\*

COPY ./public-html/ /usr/local/apache2/htdocs/

Add a /public-html dir to the image with an index.html file

docker build -t mjrw/httpd-ex .

Docker docker run --name httpddemo -p 81:80 –rm mjrw/httpd-ex

Run the image – map port to 80:80

\*/

docker exec -it sad\_franklin /bin/bash

Explore

# Defining a Service Interface with Swagger

Create a service interface

swagger.io

Select tools / editor – online editor

Create a swagger specification for a Member service

Use the heoroku pet store as a template

Methods

get given and id returns an object representing a member

name

address

email

post adds a new member

put updates a member

delete deletes a member

# Service Aggregationlocalhost:8080

## **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:
  + {   
     “memberId”: 0,  
     “memberName”: “abcd”,  
     “email”: “x@y.com” “purchases”: [  
     “subscriptionId”:0,  
     “subscriptionName”: “abcd”,  
     “price”: 0.0  
     ]  
    }

## **Major Steps**

* **Create an OpenAPI definition for the aggregating services**
* **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**

**Create an OpenAPI definition for the aggregating services**

* Using the Swagger tooling, create a new definition for a the Member-Subscription-Purchase service.
  + To keep things simple the service should just support a Get operation. The operation should take a single parameter of memberId (an integer).
  + The service should return a data structure as show in the outline (above) which includes an array of objects with details of the purchases.

**Generate a service stub**

* You now have a choice: you can perform the remainder of the exercise with JavaScript / NodeJS for which we have provided a framework or you can be more ambitious and create the service with your preferred platform (though you need to check that that Swagger supports it). Our instructions focus on the JavaScript NodeJS approach.
* 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 adocs/#!/default/byMemberMemberIdGetggregate 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.

# 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 it’s capabilities.

## **Detailed Steps**

* Start the Docker container with the command

docker run --name mongo-container -v /home/mjrw/Documents/BBC-Microservices/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/)

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

db.subs.find({"Member\_Name" : /Michael.\*/});

Do some sample queries (see 4500)

Give them a query to figure out

\*\*\* Integrate with Node?

# CASSANDRA

docker run --name some-cassandra -v /home/mjrw/Documents/BBC-Microservices/Resources:/mydata -d cassandra

docker exec -it some-cassandra bash

cqlsh

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

CREATE KEYSPACE subscriptions WITH REPLICATION = { 'class' :

'NetworkTopologyStrategy', 'datacenter1' : 2 };

(Network replicated)

use subscriptions;

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

COPY crimedata (id , name , member\_id , type , renew , startdate , enddate )

FROM '/mydata/subscriptions.csv;

Neo4J

docker run \

--publish=7474:7474 --publish=7687:7687 \

--volume=$HOME/neo4j/data:/data \

neo4j

Connect to 7474:7474

Run the code sample

Docker Swarm

Connect Swarm to AWS

(Use pre-created AWS role).

Create Swarm – show resources being created

Connect to the swarm

Deploy the Composition onto the Swarm

\*\* Connect to a node and see what’s started?

db.testdata.find()

# Membership Application

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