**Introduction**

This documents charts my thoughts/decisions etc. as I have completed this project. It is broken into two sections, one documenting my development journey, with the other documenting my efforts to deploy my project to a cloud service.

**Choosing a Project**

Initially I was thinking of writing some generic service (e.g. a hotel reservation system or a fantasy football league) in C# using the ASP.NET Web API 2 or the ASP.NET Core framework. I was planning on using Azure to get these hosted, and was planning on using React.js as the framework for the user interface. These are the technologies I’ve most worked with and am familiar with.

However, during Assignment 1, I enjoyed being exposed to the technologies involved there which were mostly new to me (express etc.), so I decided to iterate on the Assignment 1 codebase.

**My Initial High Level Plan**

* Firstly, I plan getting my environment setup so that I can deploy to Bluemix or AWS. This is something I’m not familiar with at all, so I want to do this in parallel with starting development.
* I want to complete the UI clean-up I started with in Assignment 1. I am using bootstrap[[1]](#footnote-1) for this.
* Add support for an Admin user – only show pages like Add Product to admins.
* Ensure the login/logout functionality works (currently it does not work correctly).
* Implement checkout feature.
  + Requires an orders service to be implemented.
* Stock Management Service – ability to increment/decrement stock levels etc.
* Catalog Service will require some additions – activate/deactivate products, delete products.
* Code clean-up and commenting.

**Development Efforts Journal**

**User Interface Clean-up**

In Assignment 1, I had added bootstrap and done some basic styling of the user interface. To complete this within the front-end service, I completed the following:

* Moved the inline CSS in Index.html to its own style.css file.
* Used bootstrap to create a new header and navigation bar integrated into the header.
* Created a new login/registration drop down in the navigation bar.
* Added consistent styling to all tables.
* Added a new Site Administration menu in the navigation bar (menu for stock management etc.)

**Add Support for an Admin User**

The idea here is that we can have different user types for customers and employees who may need to update products, add products etc.

To achieve this, I am going to do the following:

* Rename the ‘customer’ table to ‘users’.
* Add a new column in this database indicating user type. This will be an integer value. 1 will be used to identify employees, and 2 will be used to identify customers.
* I’ll then need to modify any code that interacts with this table and ensure that it works successfully.
* I’ll then review all code that is used to access pages etc. so I can use the user type to determine if a user should be allowed to access that page.

I added some new cookies, one called user\_type and one called customer\_id that I use in the application.

For example, I use user\_type to determine wheatear or not to show certain items in the UI (e.g. the admin pages) to the user.

**Creating the Order Service**

As I reviewed the Assignment 1 codebase in terms of what was missing for me to create a functioning application, it was obvious that in order to implement features like checkout, I would need to implement an orders service which would be responsible for processing user orders.

I started by create a skeleton orders service with no functionality that would run on port 3005 (locally).

Next in the front-end service, I created express middleware functions in api/orders/index.js in order to be able to call the orders service.

Now I need to implement the actual functionality of the orders service. I started with an API (located at /order) where order details could be posted. This API would take care of updating the database:

* Update the orders table with the high level order details (customer ID, date etc.)
* Update the orderdetails table with the details of the items actually being ordered (tied back to the relevant row in orders).

Also when the order logic is called, the stock levels of the relevant stock being ordered should be decremented.

While implementing the orders service, I also discovered a defect in the user service.

The user service was not checking the password provided, only the username. So as long as a valid username was input, the login would always be successful, even if the password for that user was incorrect. I modified the user service to take account of the password also.

**Checkout**

I decided to create a very simple checkout flow, one that wouldn’t involve payment as at this point I was worried about time because of how my efforts trying to get the services deployed to AWS were going (see later in this document).

I implemented the checkout flow such that the following happens when a user checks out from the cart:

* I get all the relevant information of the items in the cart (via the catalogue service).
* Using the orders service I add details of the orders in the database and give it an initial state.
* I message the user via a modal indicating if the order has been placed successfully or not.

There is room to expand this functionality perhaps later if I get time to come back.

**Creating the Stock Service**

Next up I created the stock service, starting with the ability to increment and decrement stock levels for particular products.

I created a new skeleton server called ‘stock’ and added the necessary code here to create these functions.

I tested these new functions to ensure expected behaviour by running the stock service and calling into each function with a REST client directly (I like Advanced Rest Client[[2]](#footnote-2)).

Also in the front-end service I added a new file at api/stock/index.js that would contain middleware functions for accessing each function of the new stock service.

Now that I had the ability to increment/decrement stock levels, I went back and integrated these into the checkout flow, so for example when a user checks out with an item of stock, the stock level is decremented.

More functions were required in the stock service, especially to build the stock management user interface I had planned in the administration panel.

I next added a function to give me an overview of all the current stock that would be used in the generation of this UI.

**Stock Administration User Interface**

Next I went back to the front-end service and created the UI for the stock administration service which would allow administrator users only to add to stock, decrement stock numbers, deactivate a product on the site, or reactivate a product that was previously deactivated.

This UI shows each product and its current stock level etc. via the API provided in the stock management service.

It also shows the current value of the stock, which is increased/reduced as stock is added/removed.

Product can also be completely deleted from here – this is an irreversible action that deletes the relevant row from the products table in the database (as opposed to just setting the active flag to 0).

**Ability to View Orders**

The user should be able to view their current/historical orders. I added a menu item in the navigation bar (only seen when a user is logged in), that gives the user the ability to view their orders.

To achieve this I needed to add a function in the orders service that would grab the orders for a particular user ID.

I also added an item in the administration panel so admins can see al orders. This required the addition of another function in the orders service to grab the data.

**Deployment Efforts Journal**

My high level plan on deployment is as follows:

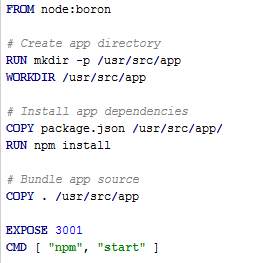
* Come up to speed on Docker in order to containerize my services.
* Investigate AWS as a means of hosting my services.
* Investigate IBM Bluemix as a means of hosting my services.

The rest of this section outlines my efforts to containerize my services and deploy them to a cloud service.

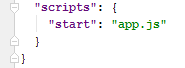
**Creating a Dockerfile & Docker Image for each Service**

As I was not familiar with Docker, I decided to take a ‘baby steps’ approach and begin by creating a Dockerfile for one of my web services, the user service. I used the online Docker documentation to come up to an initial level of understanding on Docker, as well as using the notes from Lecture 10.

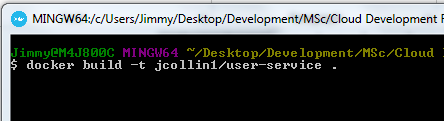
For the user’s service, my Docker file looks like this:



Note that I had to make a change to package.json in the user service also, due to the main file being called ‘app.js’ and not ‘server.js’ in this service:



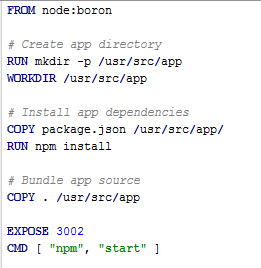
Once I had my Dockerfile created, I used Docker for Windows to create a Docker image using the following command:



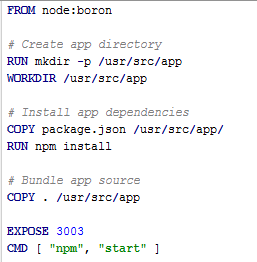
Now I had my user service Docker image, so I decided to create one for each of the required services, including the new services I was working on (the Stock Management service, and the Orders service).

Below are the Dockerfiles I created for each of these.

*Catalogue Service*



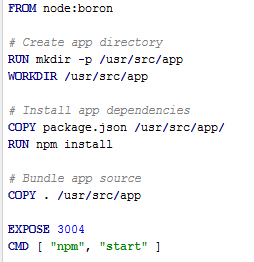
*Cart Service*



*Front-end Service*



*Stock Management Service*



*Orders Service*



Now that I had a Docker file for each service, and an image created, the next step was to test out actually running the images as Docker containers on my local machine.

To do this I used the below command (in this instance for the front-end service) to get each container running:



The full list of commands I used to get the services running is:

* docker run -p 3001:3001 -d jimmyc/user-service
* docker run -p 3003:3003 -d jimmyc/cart-service
* docker run -p 3004:3004 -d jimmyc/stock-service
* docker run -p 3005:3005 -d jimmyc/order-service
* docker run -p 8079:8079 -d jimmyc/front-end
* docker run -p 3002:3002 -d jimmyc/catalogue-service

***Note****: About this time I had a conversation with someone about Docker Compose, and decided to investigate it later. My priority for now was to make sure I could containerize each service and get them running successfully on my local machine.*

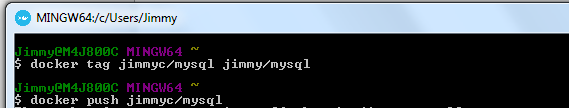
I had a couple of problems with dependencies during the process of getting my services up and running in Docker containers. This was due to the fact that the Assignment 1 codebase has a ‘*node\_modules*’ folder in the source tree, but the latest version of these dependencies are grabbed when the node app is being created in the Docker container.

Whenever a service was acting up or not behaving as expected, I started that container last, and without the ‘-d’ parameter in the ‘docker run’ command. This allowed me to see the stack traces for any issues that happened.

Another problem I’m having is that most of the services that interact with the database have a hard coded localhost variable pointing at the database – how to handle this locally (by hardcoding?) or in an environment like AWS or IBM Bluemix?

After I had created each of my Docker images, I decided to create an account on DockerHub and upload my images here as a backup.

To do this, I used the following commands to create a tag and push each image:

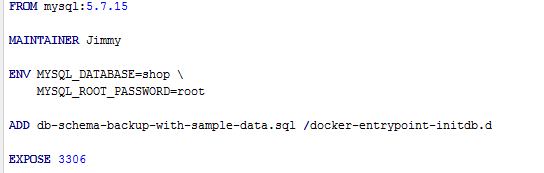


These can be viewed at <https://hub.docker.com/u/jimmyc/>.

**Creating Dockerfile and Image for MySQL Database**

At this point my services were running fine in Docker containers, but without the MySQL database they would not be much good.

To create this, I created the below Dockerfile.



After some trial and error, the above can be run using the below command and it creates the MySQL container and also runs the necessary SQL (stored in the db folder in the source tree) to setup the required tables in the database.

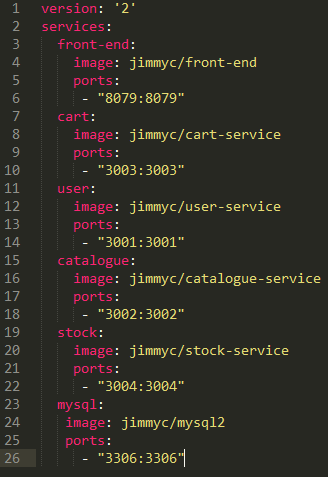


**Utilizing Docker Compose**

As mentioned earlier, I had a conversation with a colleague who had recommend that I look at Docker Compose as a means of defining and running my multi-container Docker application.

Through some research and reading of the Compose documentation[[3]](#footnote-3), I was able to get a simple Docker compose file up-and-running that creates the necessary containers for my application in one Docker Compose command (docker-compose up).

My Docker Compose YML file is shown below. This is committed to the root of the source tree also.



**Attempts at Deploying to Amazon Web Services**

**NOTE**: I had initially decided to try to deploy to AWS, but midway through I decided to back-out of this due to two things:

* I decided that the complexity and learning-curve of AWS were too much in the time I had to do this assignment (especially as I had a lot of code to write also).
* I started to get charged for my AWS usage (only $10.01 albeit) – I had signed up for the free tier usage, so I’m not sure what happened, but when I saw this and my next month projected bill of $133, I immediately cancelled my account.

I have included the steps I had taken on AWS here anyway.

I learnt that you can deploy Docker containers using the Amazon EC2 Container Service[[4]](#footnote-4).

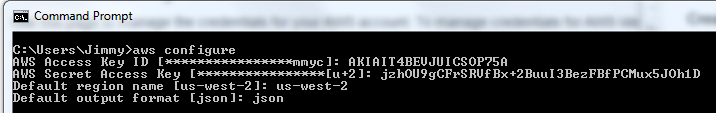
The first thing I done was to sign up for the AWS Free Tier (or so I thought).

Next, I got setup in ECR as per the documentation[[5]](#footnote-5).

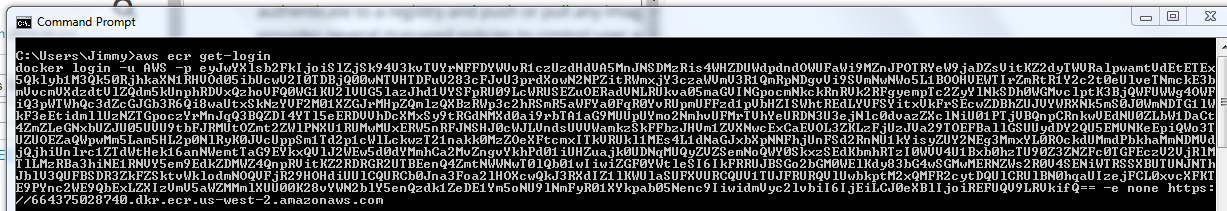
In order to be able to push Docker images to AWS, I would need to install and configure the AWS Command Line Interface (CLI).

I installed this on my Windows machine and created a new access key via my Security Credentials dashboard in AWS.

After installation, I next ran aws configure from my command line to configure the CLI:



This gives us the command we need later to authenticate our Docker client with the Amazon ECR registry:



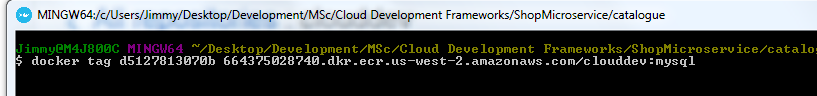
(Authentication tokens are valid for 12 hours)

Now my AWS CLI was operational, so following the instructions here[[6]](#footnote-6) I begun by creating a new repository to store my Docker images.

Next, using Docker I authenticated with the repository I had created in AWS:



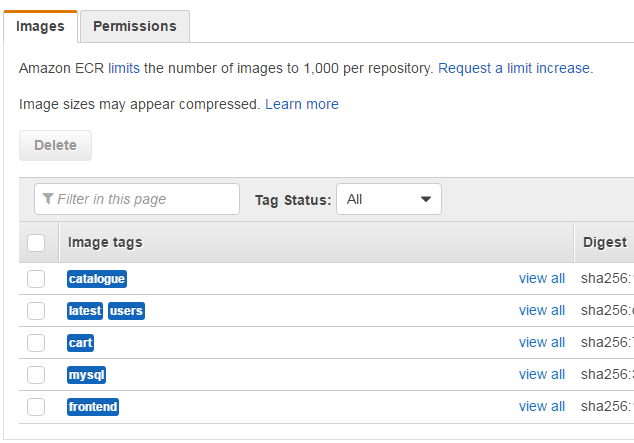
I then tagged each of my images for deployment to EC2, e.g.



Finally, I pushed each of my images to the repository in AWS:



Now I could see my images in the repository in AWS:

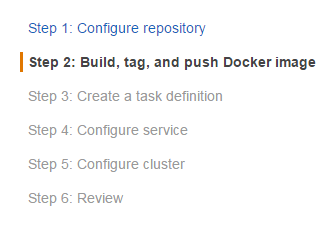


Now that my images were deployed, the next thing I needed to do as per the AWS documentation was to create a task definition[[7]](#footnote-7).

“*A task definition is like a blueprint for your application. In this step, you will specify a task definition so Amazon ECS knows which Docker image to use for containers, how many containers to use in the task, and the resource allocation for each container.*”

**NOTE**: Soon after this I began to start getting charged for AWS usage, so I closed my account.

It looks like I was on Step 3/6 in terms of getting my services up and running on AWS:



**Attempts at Deploying to IBM Bluemix**

After my failed attempts to get my application up on AWS, I decided to look at IBM Bluemix, the recommended solution for getting our final projects deployed to a cloud service.

First off, I signed up for IBM Bluemix using my CIT credentials in order to get a free trial (hoping that it wouldn’t end up like my ‘free tier’ on AWS).

I then followed the instructions[[8]](#footnote-8) to install the Cloud Foundry CLI and Bluemix CLI tools on my Windows machine.

Firstly I decided to try getting my docker-ized services deployed to Bluemix.

After completing the above steps, I ended up with a private container repository on IBM Bluemix, located at registry.ng.bluemix.net/jimmyc.

First I needed to authenticate correctly with Bluemix to push images. From the Docker CMD, I ran the following commands:

bluemix login (login with Bluemix credentials).

bluemix ic init

Next, to upload my images to this repository, I tagged each one for upload:



Then I uploaded each one individually, specifying the tag:



Now it looks like my images are on Bluemix:



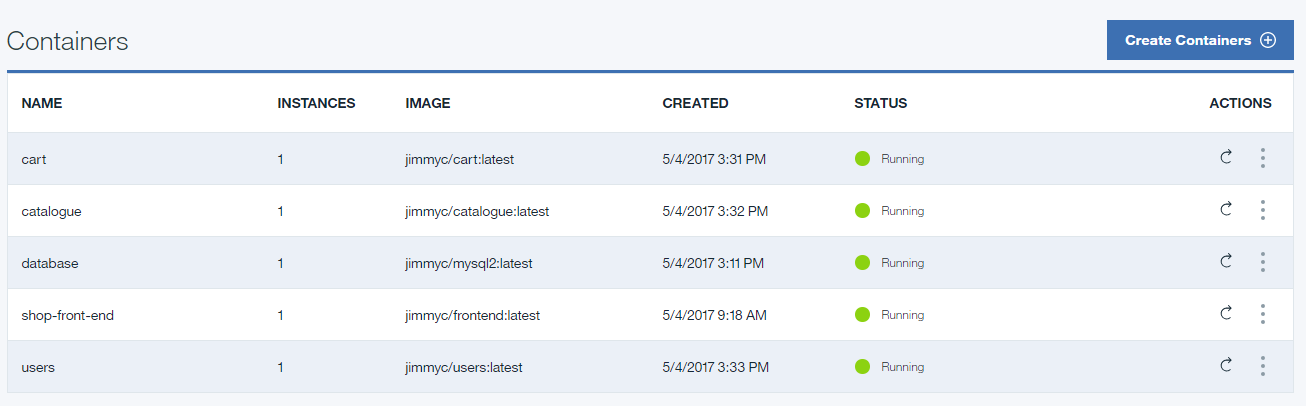
Questions I have at this point:

* I’m still concerned about how my services talk to the database (as mentioned earlier each one has a hard coded localhost variable used for the database).
* How/can I utilize Docker Compose in Bluemix?

At this point, I tested spinning up my front-end service on Bluemix to see if it would at least show the user interface (the database etc. would not work yet).

I spent a good few hours wondering why my initial attempts didn’t work, until I realised that our corporate network firewall was blocking all the traffic due to the UI being served on port 8079 – serves me right for working through lunch!

At this point, all my services are running successfully in containers in Bluemix:



The problem I now need to look at (and am not sure about) is how these services communicate with each other in Bluemix.

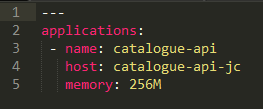
At this point (*having read lots of the Bluemix documentation and also finding some great Cloud Foundry documentation on the Pivotal Web Services site*), I decided to try to deploy my services as Cloud Foundry applications instead of creating Docker containers locally and pushing those.

As with my other deployment attempts, I decided to try and get one service up and running first. I decided on the catalogue service for two reasons:

* It would be easily testable – I would be able to use the Advanced REST Client tool to ensure that it was up and running and serving data.
* It would allow me to explore how to connect a Cloud Foundry app to a database – as I mentioned earlier I was aware of the limitations in my current code with the database being hardcoded as ‘localhost’ and having hard coded credentials, an approach that obviously won’t work in a cloud deployment.

First up I decided to create a new Cloud Foundry application. I don’t this via the user interface in Bluemix.

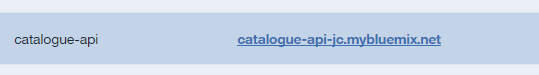
Next, I created a manifest file in my source tree catalogue folder for the catalogue service.



Now to deploy my catalogue service as a Cloud Foundry application, I followed these steps:

* Open up CMD and run cf login (I already had the Cloud Foundry CLI installed from earlier).
* CD to the catalogue directory.
* Run cf push to push the catalogue service.

This was successful and now I had my catalogue service up and running as a Cloud Foundry application, but it was not yet connected to a database.



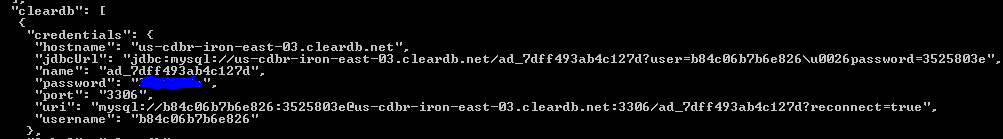
Having read lots more documentation about how I could create a database in the cloud and connect my service to it, I now had an idea of what I needed to do, and more importantly what code changes I needed to make (*something that had concerned me as an unknown for some time*).

First I created a new service in Bluemix for my database. I decided to use the ‘ClearDB’ service and got it up and running in a few clicks.



Next, via the Bluemix UI, I bound this service to my ‘catalogue-api’ application created above.

Now that this service was bound to my catalogue app, I opened up CMD and ran cf env catalogue-api to get the environment variables. This showed my the credentials to use for the database:



Next, to ensure my that my datebase was up and running, I connected to it via MySQL Workbench. I also ran the SQL to create the database schema and populate it with some test data for the user interface. (SQL is located in the db folder in the source tree).

Now I needed to make some code changes in the catalog service in order for it to be able to connect to the database locally (i.e. when I am developing), and in the cloud when the database is running in ClearDB.

In Bluemix, the VCAP\_SERVICES environment variable provides access to the services that around bound to an application, so I resolved to use this to grab the necessary database details when the application is running in the cloud.

If the VCAP\_SERVICES environement variable can’t be found, I just use localhost for the database URI and the default credentials.

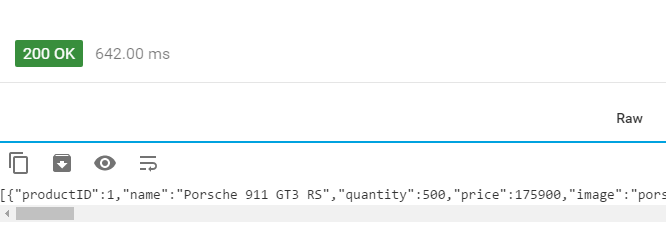
This solution allows my code to work both locally and when the service is running in the cloud.

An example of this solution can be seen on the next page.



After these changes I re-pushed the catalogue service app, and used ARC to test that I could now query the database via the catalogue service.

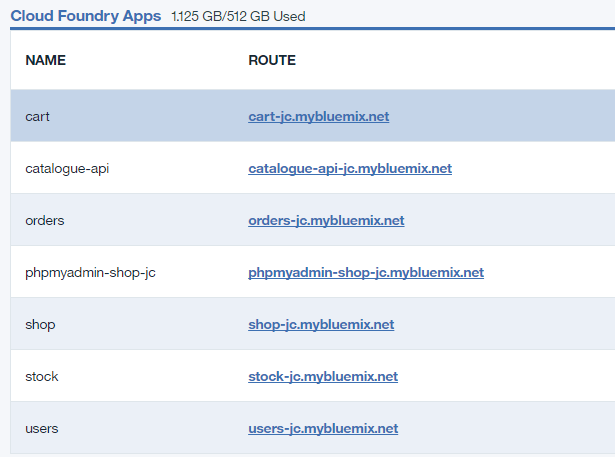
It worked perfectly:



Now that I had proven that I could deploy one of my services and connect it to a database in the cloud, I needed to go and make the same code modifications to each one to handle either connecting to a database locally or in the cloud.

Then I created Cloud Foundry applications for each one in Bluemix, a manifest file for each in the source tree, and pushed each one to Bluemix.

I also created the necessary connections to the ClearDB service (for the user, stock, and order services).





You’ll notice above that I also have a phpMyAdmin deployment. I was using this to admin my database, but found it extremely slow, so I favoured just connecting via MySQL Workbench from my local development machine as mentioned earlier.

1. <http://getbootstrap.com/> [↑](#footnote-ref-1)
2. <https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo> [↑](#footnote-ref-2)
3. <https://docs.docker.com/compose/> [↑](#footnote-ref-3)
4. <https://us-west-2.console.aws.amazon.com/ecs/home?region=us-west-2#/getStarted> [↑](#footnote-ref-4)
5. <http://docs.aws.amazon.com/AmazonECR/latest/userguide/get-set-up-for-amazon-ecr.html> [↑](#footnote-ref-5)
6. <http://docs.aws.amazon.com/AmazonECR/latest/userguide/docker-push-ecr-image.html> [↑](#footnote-ref-6)
7. <http://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html> [↑](#footnote-ref-7)
8. <https://console.ng.bluemix.net/docs/containers/container_single_ui.html#container_gettingstarted_tutorial> [↑](#footnote-ref-8)