CAB432 Cloud Computing
Mashup/Docker Project Specification

Release Date: August 10 2019
Submission Date: Monday, September 16 2019
[Monday of Week 9]
Weighting: 30% of Unit Assessment
Task: Individual Project

Aims:

The aims of this assignment are to:

- Introduce the use of cloud based VM resources for application hosting
- Practise configurable deployment of cloud applications using the Docker container
- Provide experience in the development of lightweight applications which draw on cloud data and services
- Reinforce your understanding of web development using modern servers such as Node.js, and client side scripting in Javascript.

Introduction:

The first assessment for the Cloud Computing unit requires that you build a web application – a server-side mashup – and then use a Docker container to deploy this application to a public cloud instance. The task may be broken down as follows:

- ***Create a simple web application***
  - A mashup of services and data (we will explain this later)
  - Most of the work must be done on the server side
  - Technically an extension of the prac exercises from weeks 4 and 5
  - Most people will use node.js

- ***'Dockerise' the app***
  - Create a Docker container that hosts your app
  - Generally using a Dockerfile and `docker build`.

- ***Deploy to a public cloud***
  - *Must* deploy to a public cloud VM running Ubuntu 18.04
  - *Azure, AWS or GCP all ok*
  - You must deploy via a Docker container to get a good mark

- ***Write a report***
  - Tell us about the mashup and the services used
  - Tell us how you built it

    - Tell us how to use it

- ***Show us that it works in a demo***
    - Most of the marking will be face to face
    - The rest will be based on your code and report

We will describe the task in more detail below.

Mashups:

The notion of a mashup was introduced briefly in the Week 3 lecture, and it is essentially a combination of individual services to provide a more sophisticated outcome. Those seeking greater inspiration should take a close look at the wonderful Programmable Web site, the source of all things API on the web:

http://www.programmableweb.com/

In principle, there are few limits on the range of services or applications you may use, and you should certainly take a look at some of these examples. We will provide a list of example services that you might use, and some possible use cases. You are free to use those, but you can also come up with your own.

You should check out your ideas by manually prototyping the workflow. And you should also apply for any API keys that you need ***immediately***. These can take time.

We will talk about the mashup in more detail below, but these are the basic guidelines:

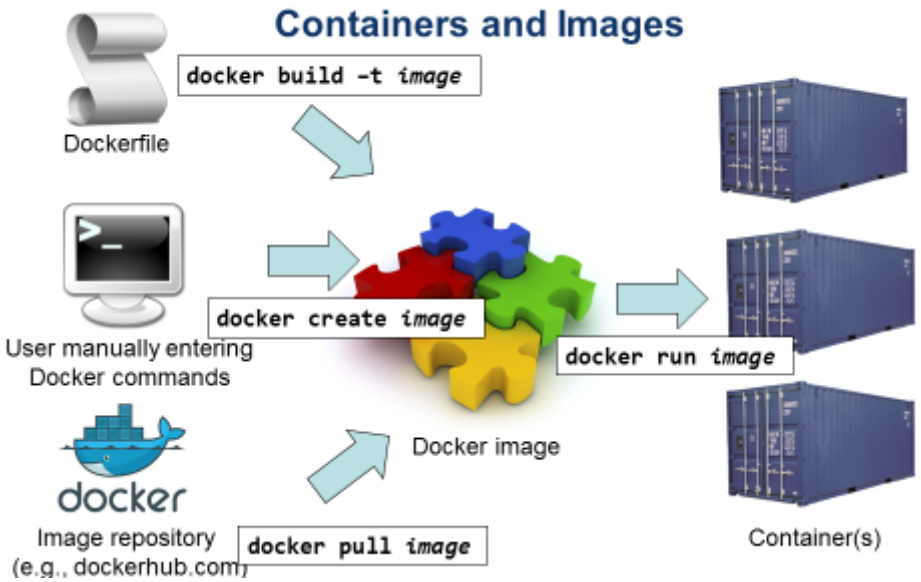
- *You cannot use existing one-click services. This clause is in place to rule out mashup frameworks, and we will leave it in just in case any still survive somewhere. Applications must be coded explicitly, and many people will find it easiest to use node.*
- Your application must be non-trivial and involve at least 2 services (details below)
- Your application must implement at least 2 distinct use cases (details below)
- Each use case ***must involve composition of the services.*** It is not sufficient to have a simple web page which displays output from two services with the same search query. See the example in the lecture presentation to clarify this. There must be some added value in the results which come back that would otherwise not have been possible.
- Your application must involve a significant server side component (details below). Some limited mashing on the client side is ok, but the bulk of the work must happen on the server, and you will find this far more convenient anyway.

Your work will be assessed according to its technical merit and usability, and on your ability to document the application and its deployment and usage (yes, we require a report). Naturally, these criteria are made more formal in the assessment rubric. There is no explicit assessment weighting given to the 'quality of the idea' as this would be very difficult to isolate, and we are providing you with some examples in any case.

The Target Environment: Docker and the VM.

You are required to deploy the app via a Docker container on top of a publicly available Linux VM. Ubuntu has been chosen because of its widespread adoption. The specific version (18.04 LTS) was chosen for consistency with the lectures and pracs and this version is mandatory. You may develop the image on a local Linux machine, a local (VirtualBox) VM, or on a cloud instance. But the final deployment must be on a major public cloud: AWS, Azure or GCP.

You have had a full lecture on Docker and Pracs in week 2 and 3 based on the use of Docker and AWS. You should all now have an AWS account and some experience of deployment, and you will find the other public clouds equally painless in provisioning VMs.



More specifically, we require that you set up (i.e., install and configure) your software stack in a Docker container. You should follow a similar strategy to the Docker prac, creating a Dockerfile which will allow you to create an image and to then deploy the container. Please pay very close attention to the port mapping.

Once the image is created, you should push your Docker image to a ***private*** DockerHub, GitHub or BitBucket repository. Deployment will then be a simple matter of pulling the image from the repo and running it on the target VM. You will need to demonstrate this deployment to us live as part of the marking process.

Other than the requirement to deploy a Linux server, there is no requirement for direct use of other cloud services for this assignment. In particular there is no need to:

- Use any cloud persistence API of any kind
- Use multiple instances, a load balancer or any autoscaling strategy.

Assignment 2, needless to say, is a very different story.

The Application:

In this section, we will give some guidance on your application. The basic guidelines were summarised above for ease of reference. Here we will go into a bit more detail.

Service or Data APIs:

- Your mashup must involve at least two (2) separate, non-trivial, service and/or data APIs. The combination must make some coherent sense and the APIs must be distinct. In the examples we give, we have organised the services into two columns, and it makes much more sense to choose one from each column.
- If you are choosing your own services, talk to the tutors or to me as a reality check - especially to check that your service choices are 'non-trivial' enough (see below). If you are struggling for inspiration, take your interests into account, and go searching at Programmable Web.

The Use Cases:

- The mashup service you provide must support at least two (2) non-trivial use cases that make sense for the domain. To make this more concrete, if my mashup focus was to provide restaurant information for tourists (say using a search API hitting TripAdvisor or some other source and a translation service and a mapping service) then a sensible use case (expressed here as a user story) might be: *As a tourist in a foreign country I wish to find a restaurant near me and be able to read their menu*.
- Broadly speaking, we will see these as separate use cases if there is a substantial shift in the underlying service accesses. So a use case in which I am seeking *Thai* food is to me NOT different from a use case in which I am seeking Italian food. But seeking Italian food from a restaurant with some minimum government hygiene rating is a different use case from just searching for Thai food. [No, I have no idea if there is a suitable hygiene API – my point is to emphasise that it would require a separate API and more parsing and processing.]

Some Constraints:

- There must be no cost to me or to QUT for use of your service – so if you wish to use an API that attracts a fee, you may do so, but it is totally at your expense. It should not be necessary in any case.
- Request your service accounts and API keys as early as possible. These may be instantaneous or they may take quite a while. Talk to us as early as possible if delays become a problem. Don't just wait until the day before submission hoping that the key will appear.

The Server:

- The actual mashing of the services – the parsing and combination – must take place largely on the server side. The architecture should be explicit in your proposal and in your final report. As noted, the software stack will be organised through a Docker container, and you will be required to deploy in our presence and to make the app accessible from a public VM. More details on this closer to the submission time.
- The choice of server is up to you, but it is intended that you use node.js and base the application on work done in the pracs. There seems no good reason to use anything other than node.js, Apache or a lightweight system like Jetty.

The Client:

- There is little alternative on the client side to Javascript, but it is not necessary to use a modern UI framework like Vue or React or Angular.
- You may base your work on a standard web page layout. You may find Twitter Bootstrap (http://getbootstrap.com/) a good starting point, or you may roll your own based on earlier sites that you have done, or through straightforward borrowing from free CSS sites available on the web. Your site should look clean and be logically organised. You will be marked down if the site is poorly designed and clumsy, but we do not expect professional graphic design. Simple is fine. Cluttered pages with blinking text reminiscent of the 1990s are not.
- If you are unsure that your front end is acceptable, do a mock up and show a tutor or me.

Please get in touch if you need to clarify any of this. In the next section we describe how you should proceed.

The Process:

The mashup project is an individual assignment and there is enormous flexibility in the specification. However it is still worth 30% of your assessment, so we need to be careful in ensuring that it is of the appropriate standard. So there will be a checkpoint to make sure that you are on track, at which time you will be given clear and unambiguous feedback on whether your proposal is up to scratch. This checkpoint will involve discussion with your tutor, and won't attract any marks, but please take the opportunity to ensure your submission will be a good one.

The checkpoint requirements below should be seen as drafts of the final report to be submitted as part of the assessment.

[Week 5]: Mandatory Proposal: During *week 5* – and certainly by sometime in week 6 - I expect you to have produced a one pager with the following information to be submitted via email to your tutor and cc'ed to me (see details below):

- Overall mashup purpose and description (1-2 paragraphs)
- At least two non-trivial use cases to be implemented. Please use the user story style: ***As a <USER-ROLE> I want the system to <DO-SOMETHING> so that <GOOD-THING-CAN-HAPPEN>***. For each one of these, you should make it clear (below) how the APIs can support the user story.
- List of service and data APIs to be utilised. This must include a short description of the API (1-2 sentences is fine), and a list of the services to be used for each user story (see above).
- A clear statement of the division between server side and client side processing, and the technologies to be deployed.
- [optional] a mock-up of your application page.
- The email should be sent with subject line [CAB432 Assignment 1 Proposal] to your tutor and cc'ed to me at j.hogan@qut.edu.au. Your tutor will look at your proposal with you in the pracs and give you feedback. The email is so that we have a record and so that I can have a quick look at the overall standard. [Your tutor will tell you their preferred email address.]

[Monday, September 9] : Report Draft: Around a week before submission, you should update your proposal and flesh out explicitly the details of the API support of the user stories, a discussion of the technical issues encountered and how they have been – or are being ☺ - overcome, test plan and results and a basic user guide. I will provide more details of the precise format in the submission guide. You need not send this to us via email, but you should use the last pracs before submission as a chance to show this to your tutor and get some feedback on the standard. As well as providing an update in preparation for submission, this will provide an opportunity to flag any projects which are not up to scratch.

[Monday, September 16] : Final Submission: This is the due date for the project. I will expect a completed report, code and tests, along with deployment and execution instructions. Precise requirements are listed below.

Submission:

The project report, source code and a copy of your Dockerfile are to be submitted via Blackboard on or before the due date. I will provide you with a submission link and detailed instructions closer to the due date.

I don't have strong views about this, but I would like to see some clear organisation of the resources, and I would also like to see some clear separation of the client and server side code. Note that it is ***NOT*** permissible to leave your Javascript and CSS in the header of the html page(s). Just ***DON'T.***

We will show you in the lectures a couple of example reports from previous years that are of the appropriate standard.

The report should be based on the proposal and liberally enhanced with screenshots. The overall length – including images and the user guide (which will mainly be screenshots) and any diagrams showing the architecture – will normally be between 5 and 10 pages.

The report should include the following sections:

1. Introduction – mashup purpose and description (updated as needed from the proposal). This should include an introductory section, outlining the overall purpose and some idea of the sorts of usages supported, but described informally (this is probably 2 or 3 paragraphs). You should then have a description of the services used, with URL and a max of one paragraph description of each. Don't get to the deep API level here.
2. Mashup use cases and services – this section should outline explicitly the use cases supported by the mashup (these *must* be illustrated using screen shots) and the service API calls used to support them. This is a semi-technical description, probably occupying a maximum of one page per use case.
3. Technical description of the application. This is a deeper discussion of the architecture, the technology used on the client and server side, any issues encountered, and overall, how you implemented the project.
4. Discussion of the use of Docker, configuration choices. You may include excerpts from the Dockerfile and list this file as an appendix to the report
5. Testing and limitations – a basic test plan, results, compromises.

6. Possible extensions – short section.
7. References
8. Appendix: brief user guide

Please get in touch if you have any questions.

Appendix: Suggested APIs

These APIs may be seen as a starting point to get you thinking, or you may use them in your mashup. As discussed above, it will in most cases make more sense to use a service from each list, and even then, not all the combinations will make sense.

Note that for each of the APIs listed, I have gone to the developer page and done a quick look to see that the API is still available, but I have not checked on the current status and timing for API keys or the reliability of the service. All of those listed here have been used successfully in the last year by my students, however. Note that some obvious choices – like TripAdvisor – are not suitable (https://developer-tripadvisor.com/content-api/request-api-access/). Some very powerful APIs such as those under Microsoft Cognitive Services may be available under an Azure subscription, but there may be costs and limits (https://docs.microsoft.com/en-us/azure/cognitive-services/cognitive-services-apis-create-account). Have a good look and ask our advice if need be.

You may also find it helpful to consider some limited use of charting libraries such as d3.js (https://d3js.org/) and chart.js (https://www.chartjs.org/) to help display some of the data, but this is not mandatory.

List 1:

1. **Flickr - images and image collections and metadata**
   https://www.flickr.com/services/developer/

2. **Songkick – API for live music information**
   https://www.songkick.com/developer

3. **Zamato – restaurant information**
   https://developers.zomato.com/api

4. **NewaAPI – news articles**
   https://newsapi.org/

5. **Twitter – Tweet streams and search**
   https://developer.twitter.com/en/docs.html
   Note that API key access can take time

List 2:

1. **Leaflet – interactive maps (JS)**
   https://leafletjs.com/
   Note that you need to choose tiles

2. **Google Maps – full mapping services**
   https://developers.google.com/maps/documentation/
   Note that Google maps requires a billing account to get an API key.
   Not usually an issue, but there are access limits and potential charges

3. **Edamam – Recipes and Nutrition**
   https://developer.edamam.com/

4. **Spotify – music metadata and streaming**
   https://developer.spotify.com/documentation/web-api/
   https://developer.spotify.com/documentation/web-api/quick-start/

5. **HostIP – geolocation from IP addresses and other services**
   http://www.hostip.info/use.html

In many cases, the ideas are immediately obvious – though I am making these up as I go along and some of them will not work all that well when you step through them manually. If the results don't seem all that interesting, they probably aren't. But let's give it a try:

- I search for a restaurant using Zamato and then I find out whether it is healthy by using their menu information to search the Edaman nutrition API.
- I take news tweets from twitter and find out more about them using the news API (yes, those are both in List 1).
- I search Spotify for artists similar to my favourite band (caveat - you will need to check that this is available in the restricted API) and then use Songkick to find information about their gigs.
- I search Flickr for location tagged images on a particular topic and display them on a map using leaflet or Google Maps (or Bing Maps https://www.microsoft.com/en-us/maps/create-a-bing-maps-key and others).
- Search NewsAPI for articles on a particular topic, and strip the URL of the published article back to the host – like www.abc.net.au. Use the HostIP API to find the IP address and then to geolocate the publisher. Then show the articles on a map. [Note, it doesn't matter for our purposes if there are errors in the location.]

These are some ideas which I have thrown together which you are free to use. Other student examples have included Zamato + Flickr + Maps (get nice pictures of the food on the menu, and optionally show them on a map), NewsAPI + Flickr (often for nature and celebrity news) and a wide range of other alternatives.

When you deal with news sources or tweets it is often necessary to parse the text. Some of this is just about stripping away JSON fields you don't need. But sometimes you want to tokenise the strings and identify the names to use as a search term. Other people have done this before ☺. You might find these libraries useful:

- CoreNLP – Stanford Core NLP for Node: https://www.npmjs.com/package/corenlp
- Natural – NLP for Node: https://www.npmjs.com/package/natural

Anyway, that should be enough to get you started. Please explore by hand and reality check your ideas, and talk to us if you aren’t sure that the approach is viable.