## SER515 Foundations of Software Engineering Extra Credit

## Containerization with Docker

One topic we did not have time to get to (but is done in SER516!) is containerization, specifically with docker. Containers are utilized to support modern approaches to build and deployment automation in a DevOps world. The short version they do this by bundling dependencies directly into a running container, so you can use the same build environment from your development workstation all the way to production. In this extra credit you will start with the basics, run a Java cli program, then convert it to a pseudo-microservices (another big benefit of docker) architecture. *Note I am not teaching this in class nor will have a coding session on it; you are required to work independently by reviewing online tutorials. I will not be taking questions in class.*

Note with extra credit we grade each task as all-or-nothing with few opportunities for extra credit. So, you have to get it just right, so follow instructions and validate your work carefully! I also do not accept grade appeals for extra credit. You will submit a Word document (Part I) which should also serve as your Readme (anything we need to know, although this should be minimal – you should ensure your tasks “just work”). The various files (Word docs and zipfiles) asked for by each Part should be combined into one zipfile named ser515\_<asurite>\_dockerec.zip (no 7zip or other formats please). *I would like to remind you of the importance of following instructions and submitting on time. Do not “forget” items; do not leave yourself enough time – don’t give away points!*

This extra credit is applied to the *assignments and quiz portions of your grade.* You can only earn up to the max points (560, see the syllabus) across the 4 assignments, 4 quizzes and the extra credit offered (the 15 points offered on assignment 3 plus this extra credit). Finally, you we will not grade your EC if you do not submit an Assignment 4 with all parts of that assignment completed.

**PART I: Run a docker tutorial (5 points)**

There are a lot of docker tutorials out there, including the docker-101 tutorial in the docker playground and the docker beginner’s tutorial from Docker. These are fine and I welcome you to run throughthem (the docker playground is a nice training tool for you to practice the docker cli without having to startup docker on your host). But for this extra credit, I want you to run the docker tutorial at <https://stackify.com/docker-tutorial/>. This one is very straightforward. Take screenshots of you doing the tutorial step-by-step for these steps on that page:

1. Running a container
2. Under the covers
3. Reuse a container
4. Share system resources with a container
5. Stop and remove a container
6. Create a Docker image, Run a custom image
7. Create a more customized image
8. Run our Python image with Pass environment variables and Hello

Stop there, you do not have to do “Share an image” (though having a dockerhub account is useful).

**PART II: Run a command-line Java program as a Docker image (15 points)**

*Task 1 (5)*: Run your assignment 3, Part II, task b “TDD” console program as a docker container. To do this, you need to dockerize a command-line Java app, meaning a) create a small Dockerfile to build your app as an image, and 2) give the docker run command used to execute it (you can put this in your Word file, along with a screen capture of you running it). In your submission name this Dockerfile “Dockerfile\_p2t1”.

*Task 2 (10)*: Take your “Calc” function(s) for the 3 operations and factor them into an API service. The API service should run in a container, while your main program (now acting as the “client”) calls the service to obtain a result. The API service should log incoming requests to the console (shows up with the docker log command) Give the Dockerfile for this service as “Dockerfile\_p3t1” and provide the full docker cli run command to run it appropriately (note this run command will be very different – it should run in detached mode [d] and have a port mapping [-p]) in your Word doc, and also give the Java host cli command you used to run the client. Your service APi must be stateful (keeps a calculator value) but does not have to persist it to non-volatile storage (just keep it in memory). Finally provide screenshots of these running side-by-side (and working).

*Note:* Yes Task 2 asks you to create a networked service, but for now you can use docker host networking (no docker network setup required). It is your design choice how you implement the API – you can find all kinds of tutorials using Spring Boot containers to create a REST API, you can create your own simple socket-based program, use JMS – I do not care (but say what you did in the Word file), but you will have to “bake-in” your selected approach’s goodness into your docker image (your Dockerfile will get more involved). Also keep in mind that running APIs should never crash – that is even if you experience errors the API service should continue running (and stable). In other words, do not let errors/exceptions crash your service!

**PART III: A gentle introduction to microservices (55 points)**

Our Part II Task 1 example is not all that interesting, as this program is standalone, has no external dependencies (outside Java), and simply runs and exits. Task 2 gets a little more interesting as there will be more external dependencies to support your API, and your are running a networked service on a port. This task will walk you through the other significant benefit of containers, microservices. These tasks are scripted to step-by-step have you “microservice-ize” your Calc program. To do this you will need to use docker-compose, which you will need to research and do some tutorials.

*Task 1 (5)*: Do this simple docker-compose tutorial, and capture the same screenshots they do in your word doc file.

Background: Microservices are a way to “slice the cake” where you implement a single fine-grained (very small, or “micro”) service as an independently runnable process. One way to do this is to deploy each behavior in an independent service.

*Task 2 (10):* Time to microservice-ize – first try

Refactor your Calc service into 3 services: one for multiply, one for divide, and one for assign. The services need to be running in separate containers. You will need to start up these containers on different ports and make it so your client know which ports to hit for which services. Your running environment should look like this:

A diagram of a software company

Description automatically generated

Submit your task solution as a zipfile Part3\_Task2\_ASURITE.zip. Inside the zip should be the all the dockerfiles needed for the 3 services, the source code tree(s), and a readme telling us exactly how to build and run the environment. This readme must be complete and failure-proof; even a single typo that causes failure means 0 points!

*Task 3 (10):* The previous task has a problem – where is the calculator value persisted? Before you had one container so you could keep it in memory, but now you have 3 running in different memory spaces. You could pull a trick and use something like docker volumes or more complex shared memory solutions, but these are really anti-microservices and anti-container approaches. Instead, create yet another service that stores the calculator value in memory. Now each of your 3 behaviors has to use this new service to share state! This service mocks a horizontal persistence service as per this diagram:

A diagram of a software company

Description automatically generated

Submit your task solution as a zipfile Part3\_Task3\_ASURITE.zip. Inside the zip should be the all the dockerfiles needed for the 4 services, the source code tree(s), and a readme telling us exactly how to build and run the environment. This readme must be complete and failure-proof; even a single typo that causes failure means 0 points!

You may be thinking to yourself: “Self, why is Dr. Gary making us do all this, this does not seem any better!”. No if anything we’ve made things more convoluted at this point – we have to run multiple services, the client needs to know about the locations and network API interfaces of 3 separate services, and the 3 services neeed to know where the horizontal persistence service is. All this for some very simple code! Let’s try to make this a little better.

*Task 4 (10)*: Apply docker-compose to create the “docker” services. Everything inside the “docker” box in the Task 3 figure should be configured to run via docker-compose on a network named “ser515\_ecnet1”. “docker-compose up” should fire up the 4 containers and put them on this network as part of one build and deployable process. This takes care of one complexity – the need to manually fire up 4 containers and map their ports.

Submit your task solution as a zipfile Part3\_Task4\_ASURITE.zip. Inside the zip should be the all the dockerfiles and docker-compose (yaml) needed for the environment, the source code tree(s), and a readme telling us exactly how to build and run the environment. This readme must be complete and failure-proof; even a single typo that causes failure means 0 points!

*Task 5 (10):* Let’s finish the job. We still have the problem that the client has to know where multiple fine-grained services live and how to call them. Let’s go back to providing a single access point to the client. Modify your docker-compose so that it puts this routing capability (called choreography) into a service as well:

A diagram of a software application

Description automatically generated

Of course you need to modify the client so it goes back to a single service (Hint: go back to Part II Task 2). Also, modify your compose to use a network named “ser515\_ecnet2”.

Submit your task solution as a zipfile Part3\_Task5\_ASURITE.zip. Inside the zip should be the all the dockerfiles and docker-compose (yaml) needed for the environment, the source code tree(s), and a readme telling us exactly how to build and run the environment. This readme must be complete and failure-proof; even a single typo that causes failure means 0 points!

OK, *phew!* All done right?

 *But wait… there’s one more thing…*

*Task 6 (10):* Right now our environment automatically persists the multiply and divide computations. First, this violates a best practice of services-oriented architectures – try to make your APIs stateless. Second, it redundantly implements the assign functionality (it has to!). Let’s correct this by making our API gateway and *orchestrator* as shown in the figure below:

A diagram of a software application

Description automatically generated

Now our API gateway can accept expression of the form “= \* 5” which says “multiply the stored value by 5 and store that value” or or “= / 3” which means “divide the stored value by 3 and store that value”. It also modifies the semantics of expressions without the “=”; for example “\* 5” now means “multiply by 5 and return that value (without storing it in the calculator)”. Keeping in mind “= by itself is still valud, e.g. “= 9” means “store the value 9”. Effectively we have just implemented the memory button on your calculator!

Submit your task solution as a zipfile Part3\_Task6\_ASURITE.zip. Inside the zip should be the all the dockerfiles and docker-compose (yaml) needed for the environment, the source code tree(s), and a readme telling us exactly how to build and run the environment. This readme must be complete and failure-proof; even a single typo that causes failure means 0 points!

*Debrief:* docker-compose is the technology, but the fundamental concepts at play are microservices as a services-oriented architecture (SOA). Tasks 3-6 show you how to assemble microservices through the patterns of *choreography* (no conductor, the client knows how to arrange services) and orchestration (a central services helps you assemble services [API gateway]). This exercise is basically one of the assignments I give in ser516, very industry relevant and I go through in detail in that course!