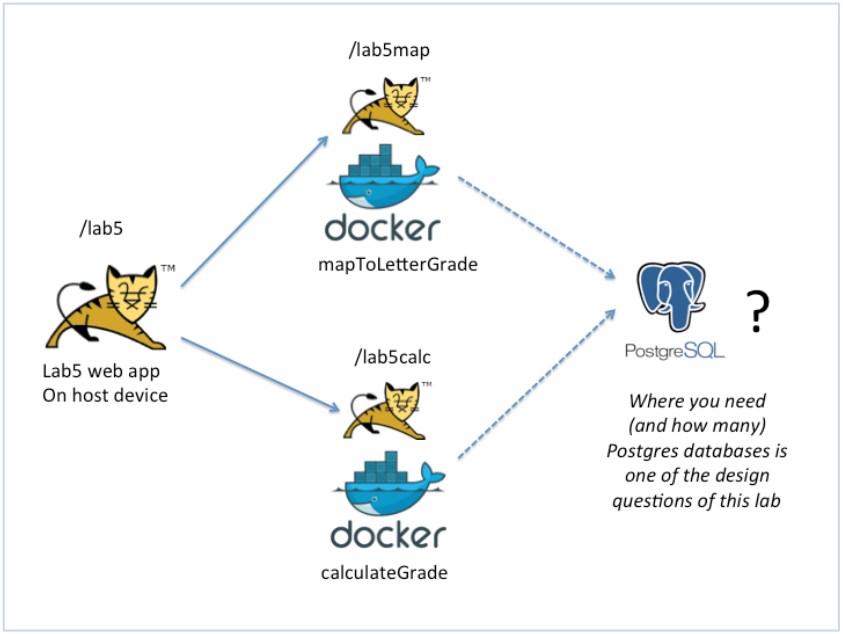
SER422 Summer 2020 ONLINE Lab 4 – Microservices

In this lab you will take an existing servlet app and refactor in-memory calls to Microservice calls. The example web application "gradeapp" has a simple servlet that uses an in-memory class GradeService to get numeric grades and mapping to letter grades. The GradeService calculates numeric scores for a mock set of Students who are in various majors (History, Engineering, English, Nursing, or Psychology) and various years of study (1=Freshman, 2=Sophomore, 3=Junior, 4=Senior). The calculateGrade method takes an int indicating a year in school, OR a String subject, OR both and returns a double value that is the average score of that cohort.
Note: *This is not a REST lab so the servlet is not written in that way. Do not worry about it!*

Your refactoring should result in the Microservices architecture shown in the figure below. Your web application should run on the host at context /gradeapp, but now the calculateGrade function should run in a Docker container at context /calc, and the mapToLetterGradeFunction should run in a Docker container at context /map. The web application should invoke each of these in turn to render the same result page that it currently does. Figuring out where the database goes is part of the assignment.

To get you started, I've created a Dockerfile from which you can build (docker build command) an image that you may use. This image runs the entire given application inside a Docker container, including the Tomcat and Postgres components. Basically, it containerizes a monolith. I am asking you to split this up, but the Dockerfile is only helpful in setting up your initial container.

Build (from directory where the Dockerfile is:
`docker build -tag lab4given:1.0 .`
Run: `docker run -p 8001:8080 <image>`
Next: `docker cp gradeapp.war <container>:/usr/local/tomcat/webapps`
Now: go to your browser on your host and enter `URL http://localhost:8001/gradeapp`



Requirements, Constraints & Grading Criteria:
Create a microservices version of the given web application by a) putting GradeService method calculateGrade into a containerized microservice, b) putting GradeService method mapToLetterGrade into a containerized microservice, and c) re-working the given web application to call the new microservices instead of using the in-memory GradeService.

1. Your architecture must resemble the figure. Your code and deployment should not rely on specific port numbers – specifically we should not have to recompile any code if the containers' ports are mapped to various port numbers of our choosing.
2. The application should be unchanged from a UI perspective (it is a simple app anyway), with the only exception that you are allowed to indicate errors if there are any problems calling the microservices.
3. Your two containers do not need to know about each other in this scenario. You do not need to use more advanced Docker capabilities like docker networking or docker compose (though see the Extra Credit).
4. Your containers should be configured to run only what they need to run (are as lightweight as possible).

Your submission README.txt must be precise and complete; this lab is as much about how the software is built and deployed as it is about the code and will be graded as such. Specifically,

1. Your README.txt must have complete step-by-step instructions on building and deploying your software to the containers.
2. If there are any manual steps we have to do, please make sure they are exactly specified where and when we run them. For example, "Run 'ant dist' from directory XXX, and then do 'docker cp ZZZ.war container:/usr/local/tomcat/webapps'".
3. Part of the design challenge of this lab is determining what to do with the Postgres database. The dashed lines in the figure represent "dependency", that is both Tomcat containers running behaviors *may* have to talk to the database. It doesn't necessarily mean another container. This is up to you! You have to decide where Postgres goes in this system.

Grading percentages:

1. Code is factored properly into microservices (verified by code inspection) 40%
   a. Proper design 25%
   b. Preserved execution correctness 5% (it is already correct, so these are free points if you don't change anything)
   c. Robust error-handling 10% (more errors may occur due to refactoring; how will you handle? Inform the user?)
2. Microservice architecture is running in docker containers 30%
   a. If not 15% partial credit for running on host under multiple servlet contexts
3. Postgres database is properly handled from a microservices standpoint 15%
   a. Sliding scale here as there are a few different design choices one can envision; one is correct, others are at different levels of correctness (cannot list without giving it away!)
4. Documentation, clarity and ease of building and running docker containers 15%

HERE ARE SOME TIPS ON HOW TO GO ABOUT THIS. NOT REQUIRED BUT HEAVILY RECOMMENDED!

First step: Ensure you can run the webapp!

1. Ensure the given gradeapp runs on your host. Yes you will need postgres if you do not have it, but you can solve this without an install by grabbing a postgres image off DockerHub. You have the sql dump so you can load a database instance.
2. Inspect the code. It is very simple, only 2 Java files. Just get a sense for what is where. Again this is not a RESTful lab or implementation, so just take the servlet and service at face value. Really simple for you at this point.

Next: Ensure Docker is working

1. Watch the video overview of Dockerfiles
2. Build the docker image from the given Dockerfile (docker build)
3. Run the image in Docker. You will need to expose ports 8080 and 5432 from the container and map back to your host at whatever target ports you desire in your run command.
4. In your web browser on your host machine, go to *http://localhost<:port_you_mapped_to>/gradeapp* and verify the web application works.

Please do these 2 things first so you know your initial environment is good to go!

Next steps: Identify your Microservices

Here are some suggestions about design considerations and how you can go about this lab incrementally.

Design

1. Recognize that since you are making network calls (yes on your one machine but simulating a physical network), there is more risk that calls will fail and your design must reflect that.
2. You will not be passing object types around through in-memory calls, so what response format will your microservices give?
3. Most importantly, what are the boundaries of your microservices? What code and what internal components are required to create each service?

Incremental Process:

1. Decide on request and response payloads. You basically are creating 2 mini-APIs for 2 methods in Lab4Service. You will have to marshal data on the wire, and the lightweight protocol of choice is JSON. Note that this specification does not lend itself to REST, so there are no constraints on the navigability or semantic metadata (or lack thereof). You just have to get information to the microservice, and read what information comes back.
    - I suggest a small console driver program or unit test to harden the format.
2. I suggest putting the containers aside until you get the microservices code organized according to your Design above. You may already have 2 running Tomcats on your workstation for this class. After refactoring the code, deploy 3 Tomcat web applications in those 2 Tomcats – the main servlet on http://localhost:8080/gradeapp (as it is now), and the other 2 on port 8081. For example, http://localhost:8081/calc and http://localhost:8081/map. Technically you can also just run them all in one Tomcat but in separate servlet contexts. Doing this keeps Docker out of the equation so you can be sure your code works first. We can give you partial credit from that if you struggle with containerization.
    - Note that this step can be broken down incrementally as well.
        1. In GradeServlet, stub out the GradeService calls to return a mock of your response payload. This will help you flesh out the format, and also consider things like error cases.
        2. Similarly, write the 2 endpoints and use a tool like Postman or ARC to send various requests with payloads and see that you get the responses you expect.
        3. In this way you can independently test each of the parts of the figure before wiring them together.
3. Once you have it working without Docker, you can then port it into Docker.
    - You can work backwards from my Dockerfile.
    - Note you can use the "docker cp" command to copy the warfiles built on your host into your containers just like the given monolith does.
4. Once you have containerized these services, then you can wire the GradeServlet to actually call these services instead of the ones on your host from above. Done!

Extra Credit: (15 points) Now put the gradeapp web application into a container as well, and use docker-compose and docker networking to wire the environment together. The network should be named lab4net and be created by the docker-compose.yml.

SUBMISSION:

- Your submission should be named lab4_<asurite>.zip for upload to Canvas. Make one zipfile with embedded zipfiles: lab4-<asurite>-app.zip, lab4-<asurite>-calc.zip and lab4-<asurite>-map.zip respectively. A README should be available in the root of the zipfile. Submit the source trees (source tree is source code, build scripts, static resources – everything required to build your software into an executable package and nothing more than that) for the web application and microservices
- Note that since you are making multiple images, you will have Dockerfiles for all of your Tomcat containers. While the given Dockerfile seems long, it is really just a concatenation of 2 files I grabbed off websites, plus a short wrapper shell script that starts up tomcats and checks on postgres and tomcat processes in the containers so you know they are healthy (in case you want to run in detached mode, which you should). Even without advanced docker and Unix knowledge I think you can figure these out and any minor changes you need. Ask questions on Slack!
- If you do the EC, you should add to the zip a file lab4_<asurite>EC.zip to the root folder of the zipfile.
- You may submit as many times as you like, there is NO reason to ask for a late submission!