Lesson 2: Science Gateway Development Environment

Bruce Becker, CSIR Meraka Institute
Sci-GaIA Winter School
bbecker@csir.co.za
Outline

- Learning goals
- Science Gateway Stack
- Toolbox
- Building and maintaining development environments
- Testing the development environment
- Practicals
- What to do when you get stuck
Learning Goals
Learning Goals

- Understand the Science Gateway Stack
  - Which components are in the stack and why
  - How they link together
- Putting our tools to work:
  - How to writing and build Docker images
  - Building functional development environments
- Testing of and in the development environment
The Science Gateway Stack
The Science Gateway Stack

In order to develop for the science gateway, we need to understand the stack of software that we will have available as underlying infrastructure.

A science gateway is – at its heart – a web portal. However, it is just not *any* web portal. Some infrastructure is implied:

- An **application server** hosts the individual components
- A **runtime environment** executes the web applications
- A **database** keeps track of user and service actions and states
- A **specialised library** provides a interface(s) to various kinds of computing and data endpoints
Science Gateway Stack – Choices, choices… standards.

There are several options available for the developer.

- **Application server / runtime environment**
  - Python/Django, Ruby/Ruby on Rails, Java, NodeJS/JavaScript *etc*

- **Specialised library** to provide interface to various kinds of computing and data endpoints. Options:
  - Write your own native interface, for each kind of endpoint
  - Use **SAGA**: A standard to govern the interface
    - Does an implementation exist for your choice above?

- **Database**
  - MySQL, PostGresSQL, Oracle … noSQL?
  - Does your database have the schema required for record keeping that target infrastructures demand?
Science Gateway Stack

- Java JRE and JDK
- Java application server (Glassfish)
- Liferay portal bundle and SDK
- GridEngine
  - jSAGA
  - Catania Science Gateways
- Database (MySQL)
Science Gateway Stack: Java JRE and JDK

The Java stack is chosen due to the existence of well-known and documented standards – particularly JSR 168 and JSR 286. These standards make your portlets portable between application servers and portals.
Several Java application servers (aka Servlet or Web container) are available - both commercial and open source. Several good Open Source options:

- Jetty
- Geronimo
- Tomcat
- JBoss
- GlassFish

We suggest (and have only tested) **GlassFish**
Your application will need to run in a website, and will need a lot of “scaffolding”. Usually the application will be used by a community or team, and will be part of a suite of tools in a workflow. If you build the science gateway website from scratch, this is a lot of work. A simple CMS will not suffice either deep functionality is needed to manage the application's (and user's) interaction with a complex distributed computing infrastructure. A “portal” is desirable – one compatible with previous choices. A good candidate to fill the portal role is Liferay.
Science Gateway Stack: “Grid” Engine

We want to be able to exploit as many computational resources as possible – but each has its own interface for:

- Authentication and authorisation
- Resource discovery
- Job submission and management
- Data movement, storage and discovery
- Accounting

jSAGA provides a Java implementation of the SAGA standard allowing a unique interface to most of this functionality, independent of the actual remote resource type.
SAGA and its respective implementation only gets you part of the way to a decent user experience. Much functionality – either required by the user agreement of the remote computing federation, or desired by the user community – has to be provided by the portal, or portlet.

A few projects have grown out of this need for “community development kits”, including the Catania Science Gateways Framework (CSGF)

CSGF provides extra Java libraries enabling richer UX and ensures compliance with EGI.eu Grid and FedCloud infrastructures, amongst others.
Science Gateway Stack: Contained
Before we move on to checking the toolbox and building the development environment, let's remind ourselves that we are here to build **portlets** for specific applications or workflows. These portlets are the last layer in the stack and will use the Science Gateway Stack, primarily the Liferay portlet SDK. You will be writing most of your code here…

… but let's not get ahead of ourselves.
Toolbox
Building the Science Gateway Stack - Toolbox

- Q: How can we build a reproducible environment to test our work and provide others a means to validating it?
  - A: Docker containers!

- Q: What is the best way of expressing our container, keeping track of how it evolves?
  - A: Dockerfile in a change-controlled repository
Good practice suggests that there should be one process per container, however we have two:
- MySQL server (db)
- Java server (web)

Separate them into two containers; Link with Docker Compose

We can now easily describe in simple text (YAML) the relationships between these two applications, in their respective containers
Building the Science Gateway Stack

The practical assignments will help you to build your development environment in a step-by-step way. Using Docker containers, we will validate each intermediate step of your work. Using proper methods, you will be able to keep control over your development environment, and if you wish experiment later with variations on the composition of the stack. Ready? … here goes!
Building and maintaining development environments
Building and maintaining development environments

1 Before you go crazy...
2 “web” image:
   1 Build the web base image
   2 Add Java
   3 Add Glassfish and Liferay
   4 Add the Grid Engine and dependencies
3 “db” image
4 Create app.yml to link them together
Step 0 : The environment

- You will need :
  - Github account
  - Docker hub account
  - Editor
  - (some automation)

- Choose a working directory
  - mkdir work

- Get to work
  - cd work
  - <edit Docker file>
Step 0 : Workflows

- Typical workflows might include a mix of working with the source code (Dockerfile) and image.
- Both the Dockerfile and the image can be committed (to Github and Docker Hub respectively)
- **Remember**:
  - We will only see the final result
  - We will compare your Dockerfile with the image in the Docker hub
- Make your Docker hub repositories **automated**
Step 0: Automated Workflows

Try to do as little as possible *yourself* to the image, let the build be done *only* by Docker Hub

1. `<edit file>`
2. `git commit`
3. `git push`
Step 0 : Semi-Automatic Workflows

Sometimes, however, this can be infeasible

1. **Edit file**
   1. docker build <name>
   2. docker run <name>

2. **Manual checks**
   1. docker commit <name>
   2. docker stop <name>

3. git commit
4. git push
Step 0 : Testing

- Remember the points on Continuous Integration from Lesson 1:
  - How will you be sure that your images are going to work?
- Try to include some tests in the repository
  - Sanity tests – are basic things as they are expected? Are you breaking any major rules?
  - Functional tests – does the application in the container actually do what it's supposed to do?
- These tests should be easily executable by Jenkins (or other testing services)
Step 1.1 : Build the web image

Use Dockerfile best practices

FROM ubuntu:trusty  # You can use apt or rpm-based distros. Which is the smallest ?
MAINTAINER Name Surname
  <name.surname@email.address.com>
# Make sure you write meaningful comments that say WHY rather than WHAT
Step 1.2 : Add Java

e.g. Oracle Java 7

RUN echo debconf shared/accepted-oracle-license-v1-1 select true | debconf-set-selections
RUN echo debconf shared/accepted-oracle-license-v1-1 seen true | debconf-set-selections

RUN apt-get update &&
    apt-get install -y --no-install-recommends software-properties-common &&
    add-apt-repository ppa:webupd8team/java &&
    apt-get update &&
    apt-get install -y --no-install-recommends oracle-java7-installer &&
    
# Can you use a different java 7 installer? Try...
Step 1.3 : Add Glassfish and Liferay

1. Download the Liferay portal “bundled with glassfish”
2. [https://www.liferay.com/downloads](https://www.liferay.com/downloads)
3. Use Version 6.1.1 Community Edition
   - Hint: [https://sourceforge.net/projects/lportal/](https://sourceforge.net/projects/lportal/)
4. Perform some necessary changes:
   1. Add glassfish server to PATH
   2. Update the glassfish admin password
   3. Start domain
5. Ensure ports are exposed:
   1. Admin port : 4848
   2. HTTP listeners : 8080, 8181
Step 1.4: Add GridEngine and dependencies

- Download GridEngine from
  - grid.ct.infn.it/../GridEngineDependencies.zip
  - grid.ct.infn.it/../jSAGA-JobManagement.jar
- Ensure that it's put in the right directory (lib directory of the liferay domain)
  - /path/to/liferay-portal-6.1.1-ce-ga2/glassfish-3.1.2/domains/domain1/lib
- Check that you have a list of jars in the lib dir.
- Don't forget to clean up
Step 2: db image

- Choose a good mysql base image
- Ensure that the service is started
- Ensure that remote connections to mysqld are possible
Congratulations, your development environment containers are ready! Now, you just need to put it all together with Compose

1. Install **Docker Compose** on your development machine
2. Create the app.yml file to build your environment
   1. Add the web and db services
   2. Ensure that the Dockerfiles are in separate directories
   3. Ensure that the correct ports are exposed *between* the containers and the outside
Testing the development environment

- Well done! You've created a development environment in two containers.
- However, will it work? Good code should provide the means and metrics for verifying its intended purpose:
  - What checks can you think of to explicitly test whether each container does what it should do and only what it should do?
  - Where should you implement these checks?
    - In the Dockerfile?
    - By hand?
    - In some automated way from your development machine?
Testing the development environment

- In any case, the tests are only as useful as the person who wrote them!
- Your code will also be independently tested by our Jenkins, providing a useful 2nd opinion.
- In practice, use as many automated test services as is feasible to show the state of your project
  - https://github.com/integrations/
Practicals
Practicals

In order to develop portlets on your laptop, you need glassfish and Liferay locally (in any location)
1. Download the Liferay bundle on your laptop
2. Download Liferay SDK on your laptop
   - Download from liferay.com
   - Check that you choose the right version
Practicals: Connect to the development environment

The deploy directory on the web container has to correspond the dist directory of the Liferay SDK

```
<path/to/sdk/>dist/
```

Hint: use VOLUME in Dockerfile

```
<path/to/liferay/>deploy
```
You are ready for Lesson 3!
What to do when you get stuck?

- We have explicitly left some work for you, and some tasks will need some discussion and thinking. Don't be discouraged by errors or problems, simply open a topic on the discussion forum or see what your peers are discussing.
- Note: We've worked hard to make this material, but we're not perfect! You may find a problem – let us know on the forum.
- Keep the Docker and other documentation handy:
  - Docker Engine User Guide
  - Docker Compose User Guide
  - YAML Reference Card
Thank you!

bbecker@csir.co.za

sci-gaia.eu