Docker and containerization

- an introduction -

INAF ICT Workshop

Milan, 21 October 2019

Stefano Alberto Russo

stefano.russo@inaf.it

Talk repository: https://github.com/sarusso/ModernSoftwareDevelopment

About me

BSc in Computer Science with a thesis on High Performance Computing at SISSA

MSc in computational physics with a thesis on Big Data at CERN

CERN research fellow working on new computing and data analysis methodologies

Then, 5 years in startups.

- Core team member of an IoT energy metering and analytics startup,
- Joined Entrepreneur First, Europe's best deep tech startup accelerator
- ..and co-founded Sharpsense (analysed the Genoa Morandi Bridge data after the collapse)

Meanwhile: always kept the link with academia.

- Lectured on Big Data for scientific computing at the Master In High Performance Computing (SISSA and UN's ICTP),
- Worked on data processing pipelines for EUMETSAT,
- Held seminars about "Modern Software Development" in universities.

Now: back into research (INAF).

A note on the startup world

Tech startups

Use new technologies

→ R&D is not the core business

Examples:

- Facebook
- AirBnB
- Amazon
- Twitter
- Uber
- ..and all the “yet another App”.

Deep Tech startups

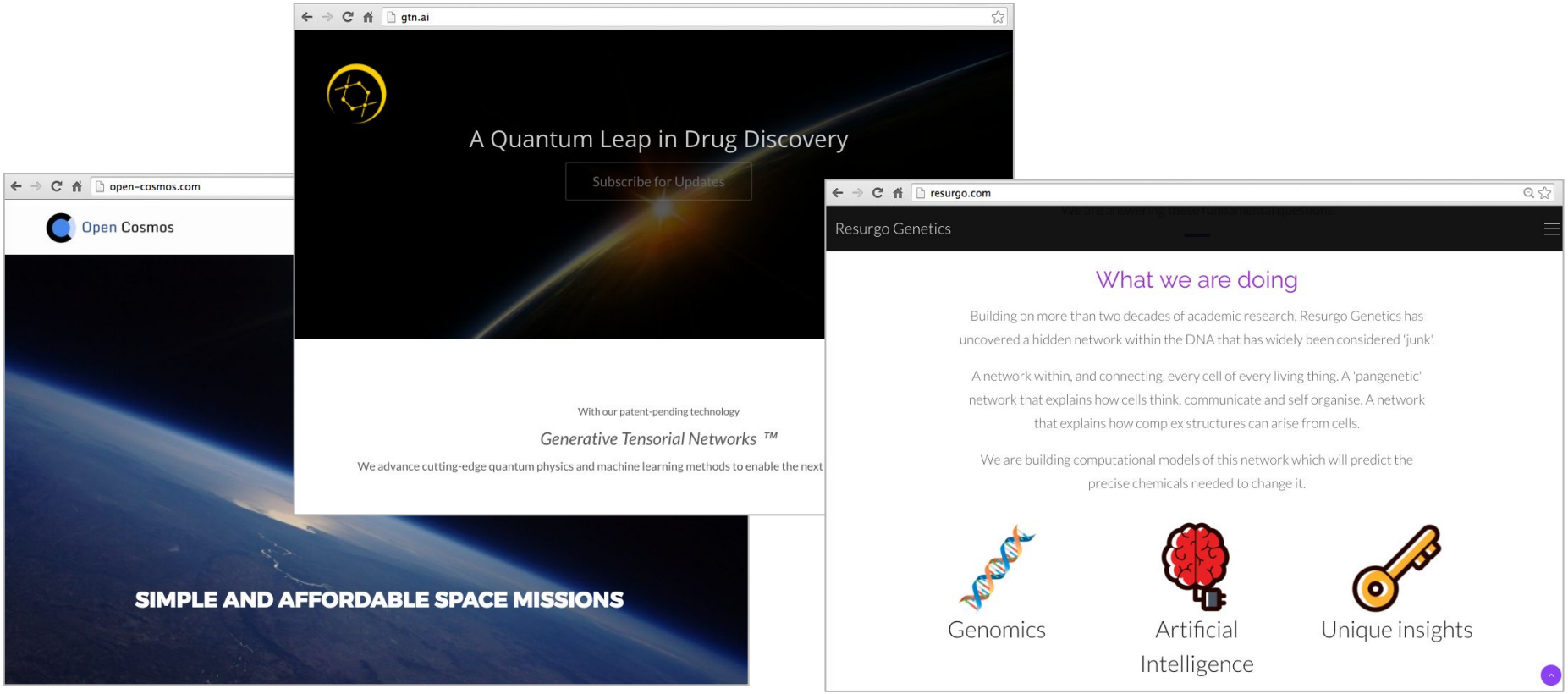
Develop new technologies

→ R&D is the core business

Examples:

- Google
- Deep Mind
- Tesla
- SpaceX
- Human Longevity
- ..and all the university spin-offs.

A note on the startup world



A note on the startup world

1) Startups are extremely resource-constrained
2) Time is a precious asset
3) Deep tech startups have complex codebases

1 + 2 + 3 = *You really don’t want to fall in a “dependency hell”*

The “dependency hell” problem

Mike wants to install a new software.

Mike cannot find a precompiled version that works with his OS and/or libraries.

Mike ask/Google for help and get some basic instructions - like “compile it”.

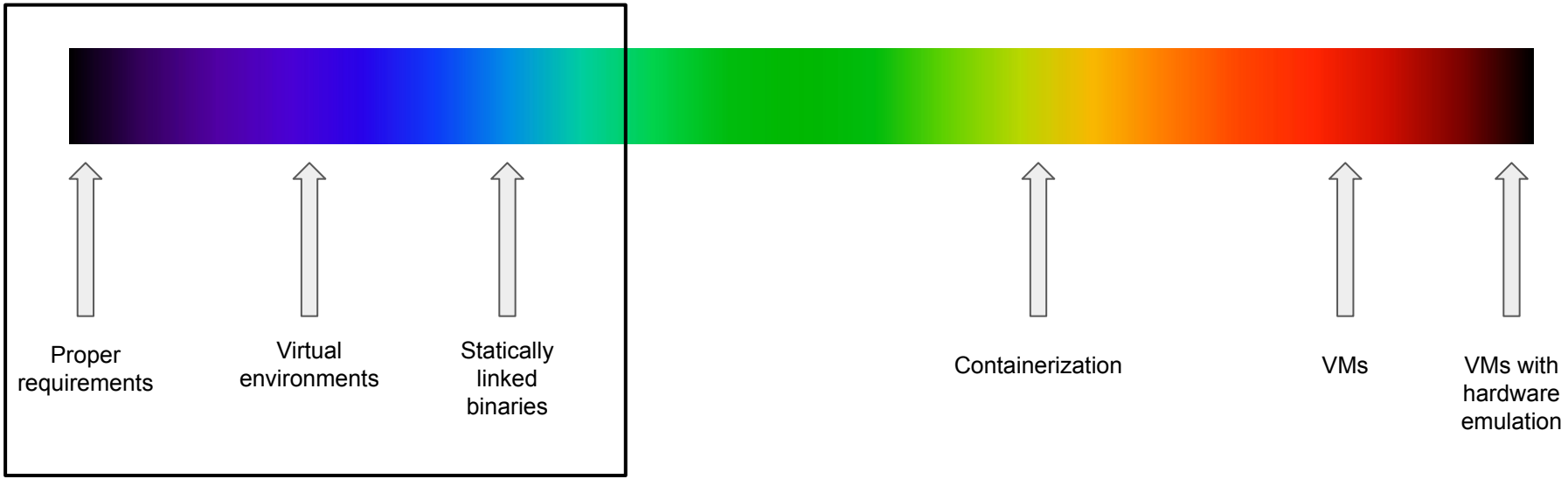
Mike starts downloading all the development environment, and soon realizes that he needs to upgrade (or downgrade!) some parts of his main Operating Systems.

During this process, something goes wrong.

Mikes spends an afternoon fixing his own OS, and all the next day in trying to compile the software. Which at the end turns out not to do what he wanted.

The “dependency hell” problem: solutions spectrum

The “dependency hell” problem: solutions spectrum



Proper requirements

- Carefully keep track of what libraries/OS features are used in development and report them on the documentation, for each release.

- Prone to human error ⇨ we stop here.

Next!

Virtual environments

- Work in a reproducible environment where libraries are the same for developers and for users. Each release has a virtual environment definition.

- Requires the user to set up and activate its own environment, and works only with some libraries (i.e. Python),

- Not a comprehensive solution and prone to human error ⇨ we stop here.

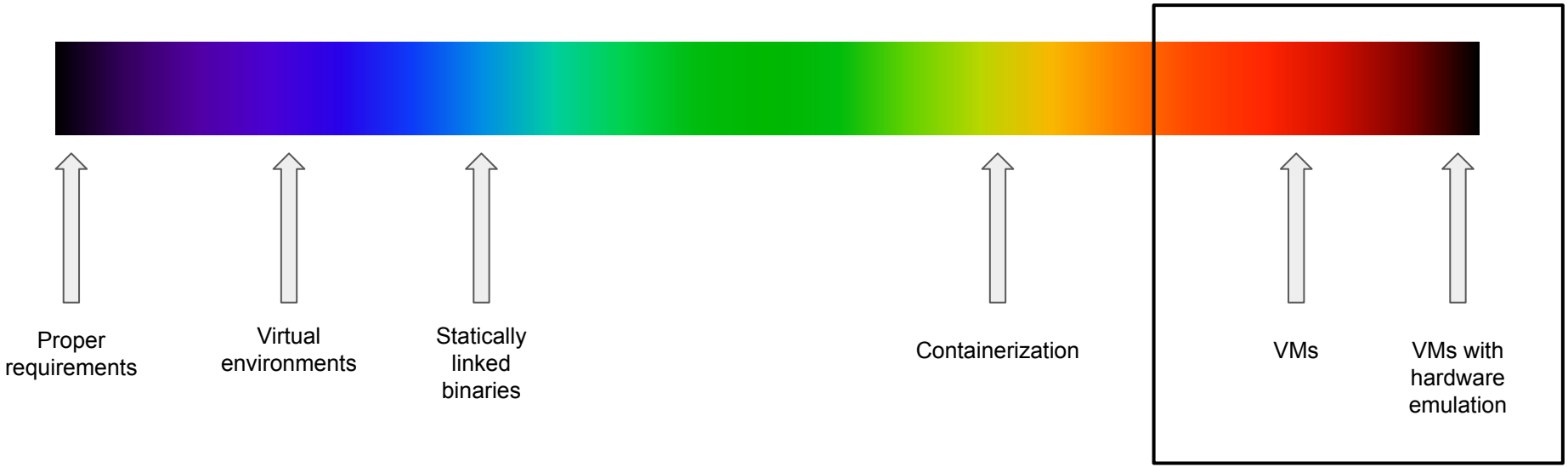
Next!

Statically linked binaries

- Works only for compiled or compilable languages ⇨ we stop here.

Next!

The “dependency hell” problem: solutions spectrum



Virtual Machines with hardware emulation

- Well… a bit of over-engineering. ⇨ we stop here.

Next!

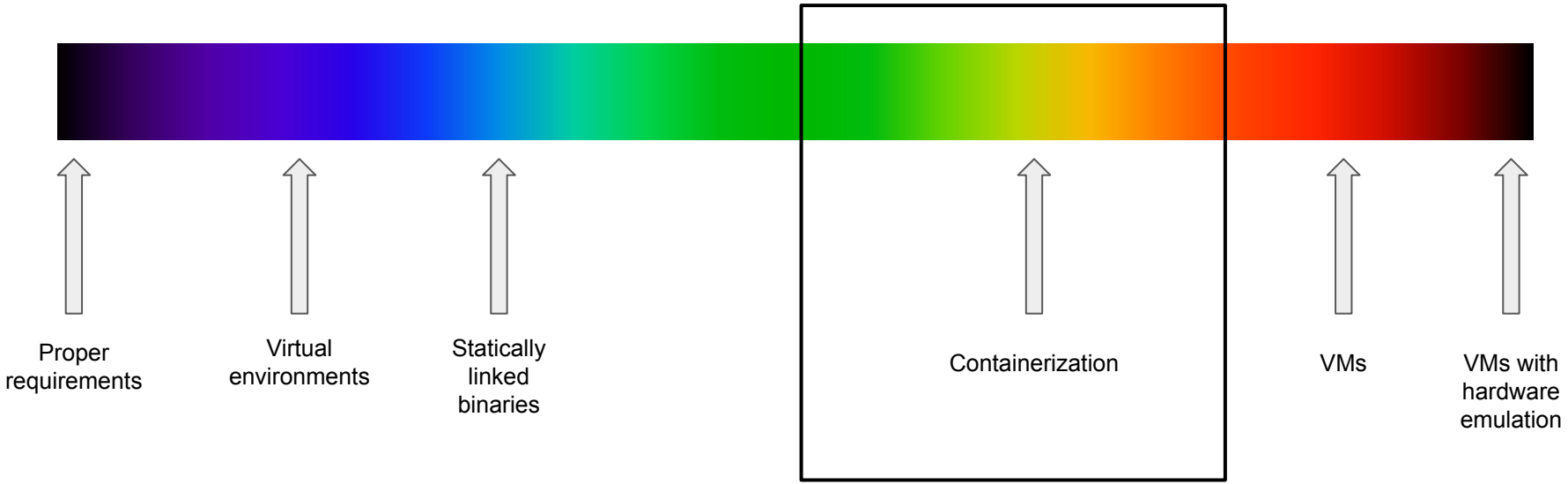
Virtual Machines

- Works out-of-the box and does not touch the main OS;
- Allows to quickly test a given software / library;
- Need to download a (big) pre-built, *trusted* image (no “source” code);
- Requires pre-allocating dedicated memory at startup, and an entire boot;
- Not suitable for much more than just giving the software a try;
- You will not find much software packaged in this way.

cons > pros ⇨ ***we stop here.***

... but we are on the right path. We want this kind of insulation!

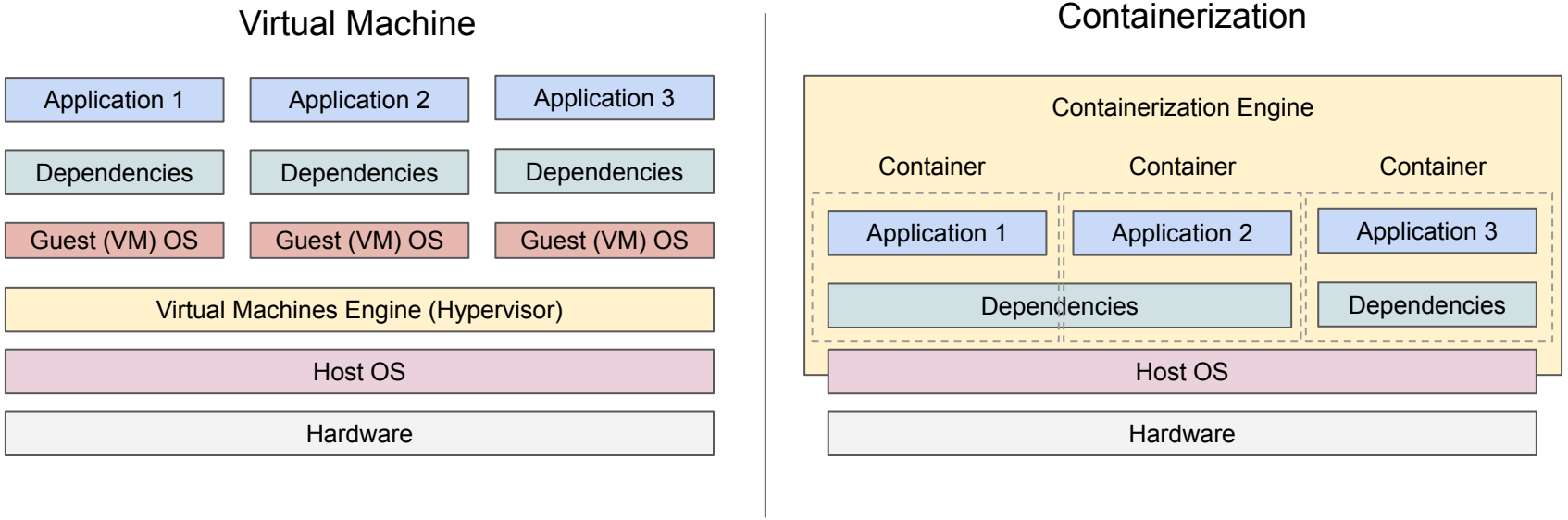
The “dependency hell” problem: solutions spectrum



Containerization

You *might* think about it as a Virtual Machine in first approximation

→ but keep in mind that they are two completely different things



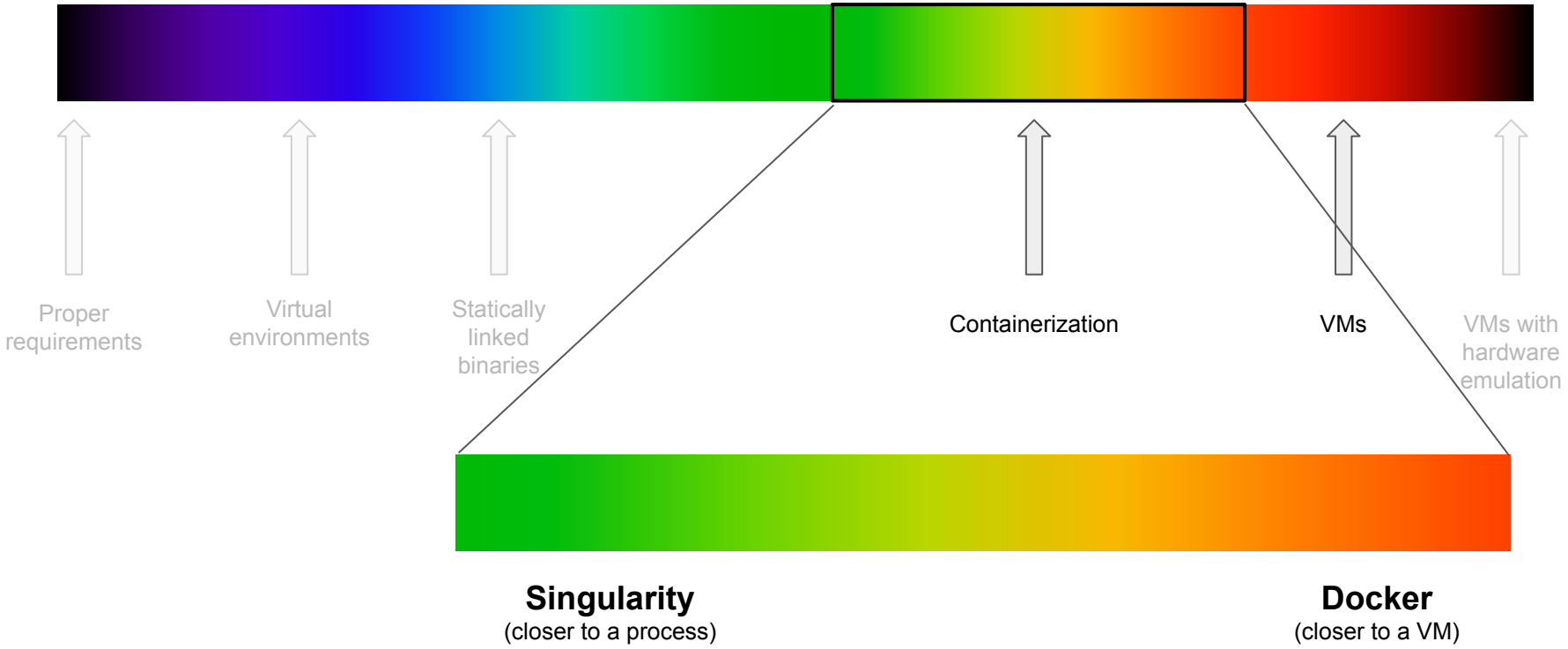
Containerization: the Virtual Machines alternative

The idea is to insulate a single process from your Operating System, and to:

- Let it live in its own space, including its own network;
- Let it have its own File System with its own libraries;
- Allow to natively access hardware without virtualization;
- Avoid booting an entire Virtual machine and to pre-allocate dedicated memory.

Different containerization solutions put more or less focus on these points

The “dependency hell” problem: solutions spectrum



Docker

- Modern containerization solution, open source
- Extremely popular, the “de facto” containerization standard
- Incremental File System
- Plenty of software on Docker Hub
- Native on Linux
- Almost native on Macs post-2011 and Windows 10 (through a light VM)

Docker Vs Singularity

Docker	Singularity
IT Industry standard	Scientific Computing
Running containers are seen as (micro)services	Running container are seen as processes
Containers have an IP address by default	Containers do not have an IP address by default
Extensive support for networking between containers	Limited or no support for networking between containers
Basically requires root access	Build as root, run as user
Limited HPC and parallelisation support	Full support for HPC use cases
Requires a daemon to orchestrate	Command line utility
Loads of orchestrators (docker-compose, kubernetes..)	First alpha stage orchestrator (singularity-compose)

Docker Vs Singularity (more technical)

Docker	Singularity
docker run: Run a command in a new container	***singularity run:*** Run the user-defined default command within a container
docker exec: Run a command in a running container	***singularity exec:*** Run a command within a container
-	***singularity instance:*** Manage containers running as services
Filesystem at runtime: completely insulated by default, use volumes to bind folders	Filesystem at runtime: only partially insulated by default, the directories $HOME, /tmp, /proc, /sys, and /dev are mounted.
Environment at runtime: from scratch	Environment at runtime: the environment of the host
Network: the one of Docker engine, use --net-host to use the host network	Network: the one of the host

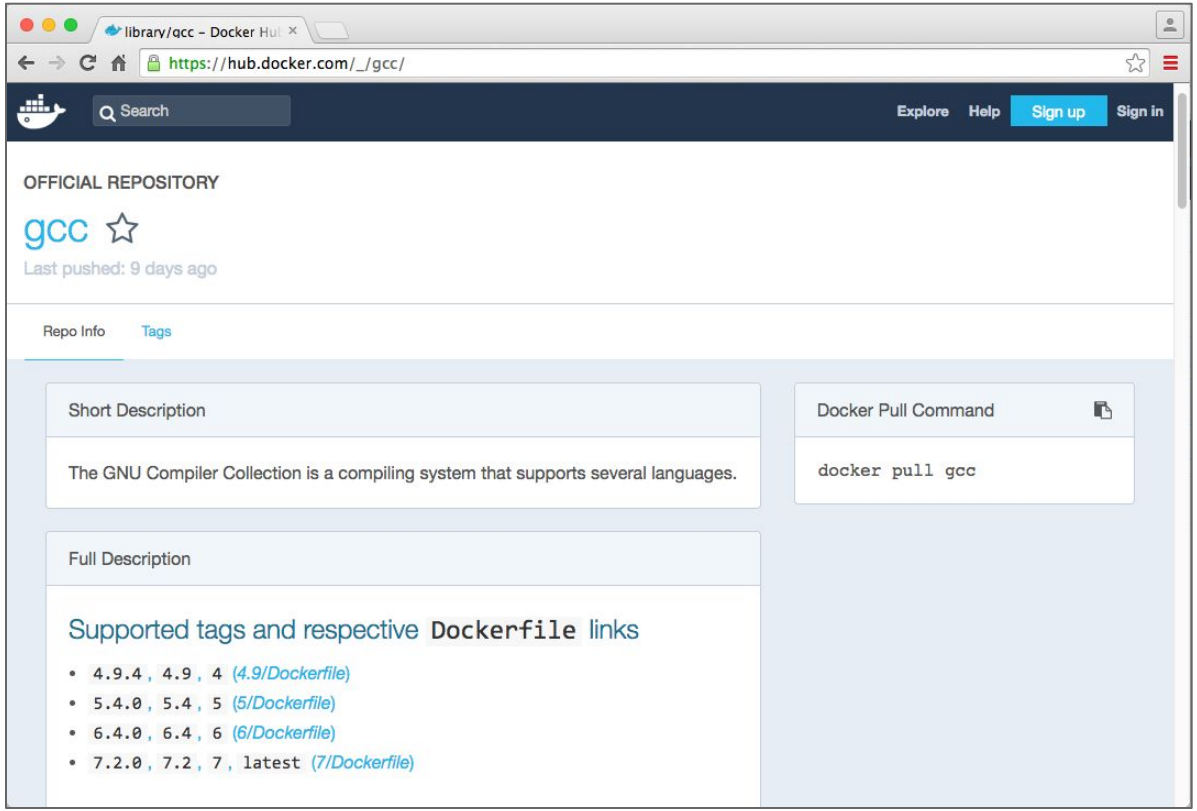
Tutorial start

We will now have a look about how to pull, run, build and share Docker containers

- You can use the `Examples/step_by_step.sh` script to follow more or less the same steps

```
$ ./step_by_step.sh

Will now pull gcc 5.4 Docker container

Command: docker pull gcc:5.4
Press enter to execute...
```

Gcc on Docker Hub



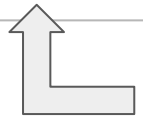
Gcc on Docker Hub (downloading)

```
$ docker pull gcc:5.4
5.4: Pulling from library/gcc
aa18ad1a0d33: Extracting [==================================>                ] 33.98 MB/52.6 MB
15a33158a136: Download complete
f67323742a64: Download complete
c4b45e832c38: Downloading [===================>                               ] 51.59 MB/134.7 MB
e5d4afe2cf59: Download complete
4c0020714917: Downloading [=======>                                           ] 30.59 MB/200.4 MB
b33e8e4a2db2: Download complete
c8dae0da33c9: Waiting
```

- You are downloading a minimalistic Linux distribution (Debian Jessie, as we will see later) on which has been installed gcc (version 5.4).
- Thanks to Docker's incremental file system, another container based on Debian Jessie *will not* require to download/store it again.

Gcc on Docker Hub (downloaded)

```
$ docker pull gcc:5.4
5.4: Pulling from library/gcc
aa18ad1a0d33: Pull complete
15a33158a136: Pull complete
f67323742a64: Pull complete
c4b45e832c38: Pull complete
e5d4afe2cf59: Pull complete
4c0020714917: Pull complete
b33e8e4a2db2: Pull complete
c8dae0da33c9: Pull complete
Digest: sha256:e6ef7f0295b9d915f8521de360e30803bf8561cfb9cea8e320aa66761be8ec42
Status: Downloaded newer image for gcc:5.4
```



Terminology warning:

- image: a “file” from which you can run a container
- container: an “entity” run from an image

Run Gcc (5.4) with Docker

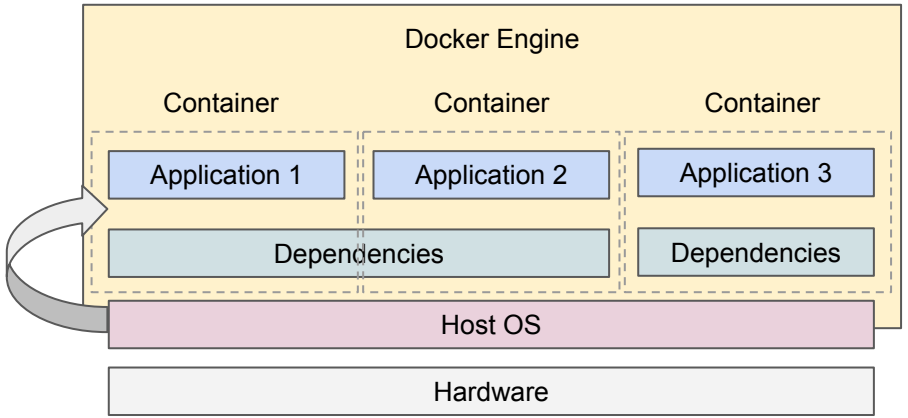
```
$ docker run gcc:5.4 gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/local/libexec/gcc/x86_64-linux-gnu/5.4.0/lto-wrapper
Target: x86_64-linux-gnu
Configured with: /usr/src/gcc/configure --build=x86_64-linux-gnu --disable-multilib
--enable-languages=c,c++,fortran,go
Thread model: posix
gcc version 5.4.0 (GCC)
$
```

Share files with a Docker container: the volumes

The container is by definition insulated from your main (host) Operating System

- But you can make some folders visible from the containers as volumes (think about an usb pendrive)
- Just append "`-v your_os_folder:path_inside_the_container`" to docker command

We are creating a bridge



Compile your code with Gcc (5.4)

Our test.c code:

```
#include<stdio.h>

int main()
{
    printf("I run a very complex simulation and the result is 42\n");
}
```

Compile your code with Gcc (5.4)

```
$ docker run -v$PWD:/data gcc:5.4 gcc -o /data/Test/test.bin --verbose /data/Test/test.c
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/local/libexec/gcc/x86_64-linux-gnu/5.4.0/lto-wrapper
Target: x86_64-linux-gnu
Configured with: /usr/src/gcc/configure --build=x86_64-linux-gnu --disable-multilib
--enable-languages=c,c++,fortran,go
Thread model: posix
gcc version 5.4.0 (GCC)
COLLECT_GCC_OPTIONS='-o' '/data/Test/test.bin' '-v' '-mtune=generic' '-march=x86-64
[...]
$
```

Run your code compiled with Gcc (5.4)

On your computer → no!

```
$ Test/test.bin
-bash: Test/test.bin: cannot execute binary file
```

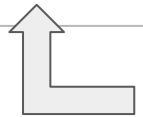
Inside the container → yes!

```
$ docker run -v$PWD:/data gcc:5.4 /data/Test/test.bin
ste@Stes-MacAir:Examples (master) $
I just ran a very complex simulation and the result is 42
```

Entering in the Gcc (5.4) container

Execute a (bash) shell in the container

```
$ docker run -t -i gcc:5.4 bash
root@b9c1414bab3d:/#
```



You are root!

List the root directories

```
root@b9c1414bab3d:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt      proc  root  run  sbin  srv
sys  tmp  usr  var
```

Entering in the Gcc (5.4) container

List running processes

```
root@b9c1414bab3d:/# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  1 13:54 pts/0    00:00:00 bash
root         8     1  0 13:54 pts/0    00:00:00 ps -ef
```

Get the container IP address

```
root@b9c1414bab3d:/# ip addr show dev eth0
    [...]
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
    [...]
```

Entering in the Gcc (5.4) container

List running Docker containers (on another shell of your computer)

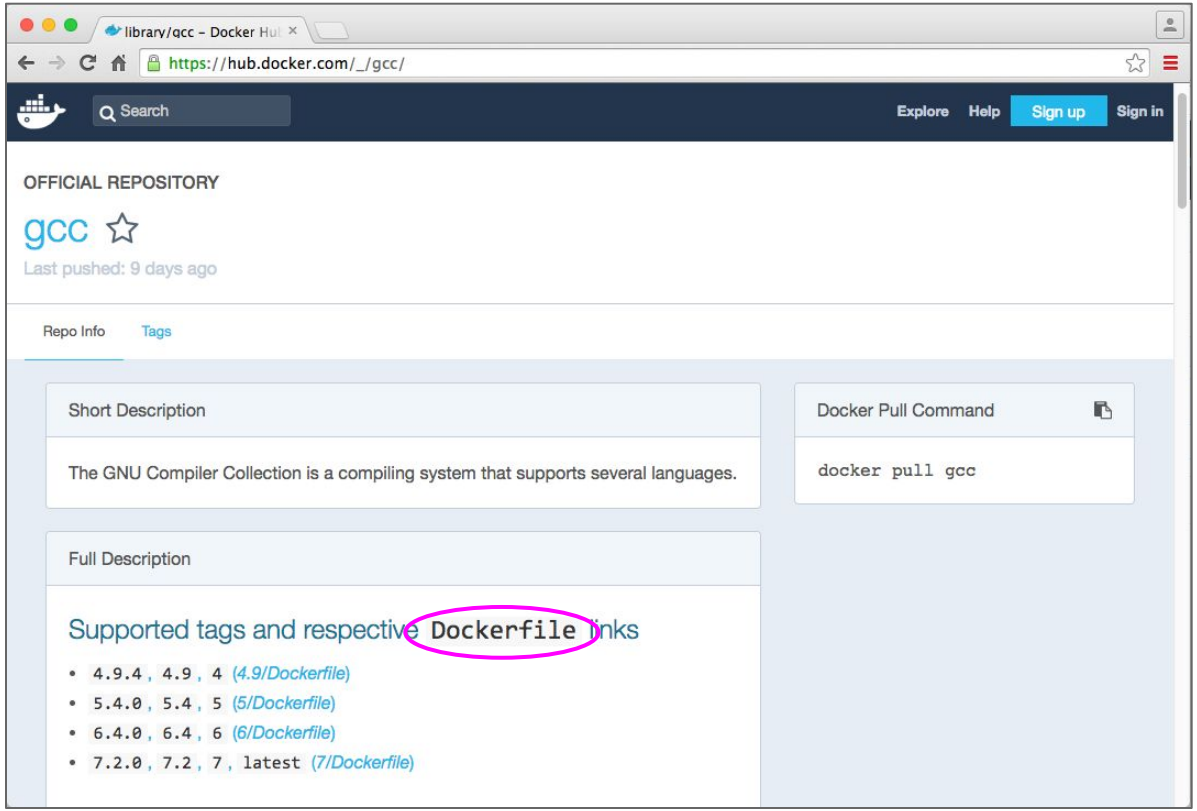
```
$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED         STATUS       PORTS         NAMES
b9c1414bab3d    gcc:5.4  "bash"    3 seconds ago  Up 1 second             friendly_goodall
```

Exit the shell, and therefore the container

```
root@b9c1414bab3d:/# exit
$
```

When you exit a container, you lose every change to the container File System

The Dockerfile



The Dockerfile

- The *Dockerfile* is what defines a Docker Container. Think about it as its source code.
- When you build it, it generates a *Docker Image*. When you run a Docker Image, this “becomes” a *Docker Container*, as mentioned before.

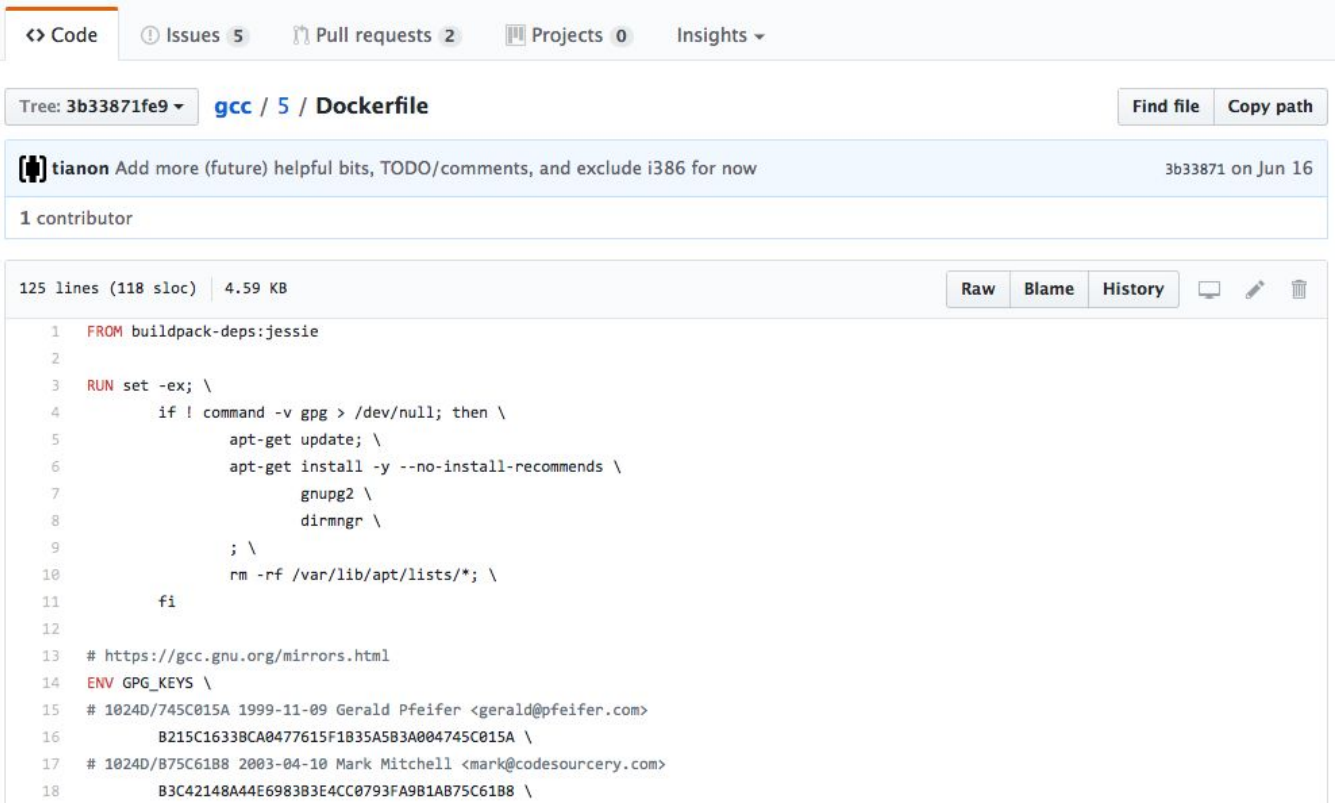
```
FROM <base image>

RUN <a setup command>

COPY <source file/folder on your OS> <dest file/folder in the container>

RUN <another setup command>
```

What is the Gcc (5.4) container built from?

There is NO black magic in Docker.

Now that we know that its source code is in the Dockerfile, we can see on what the Gcc (5.4) image is *built from*.

What is the Gcc (5.4) container built from?



Code | Issues 5 | Pull requests 2 | Projects 0 | Insights

Tree: 3b33871fe9 gcc / 5 / Dockerfile — Find file | Copy path

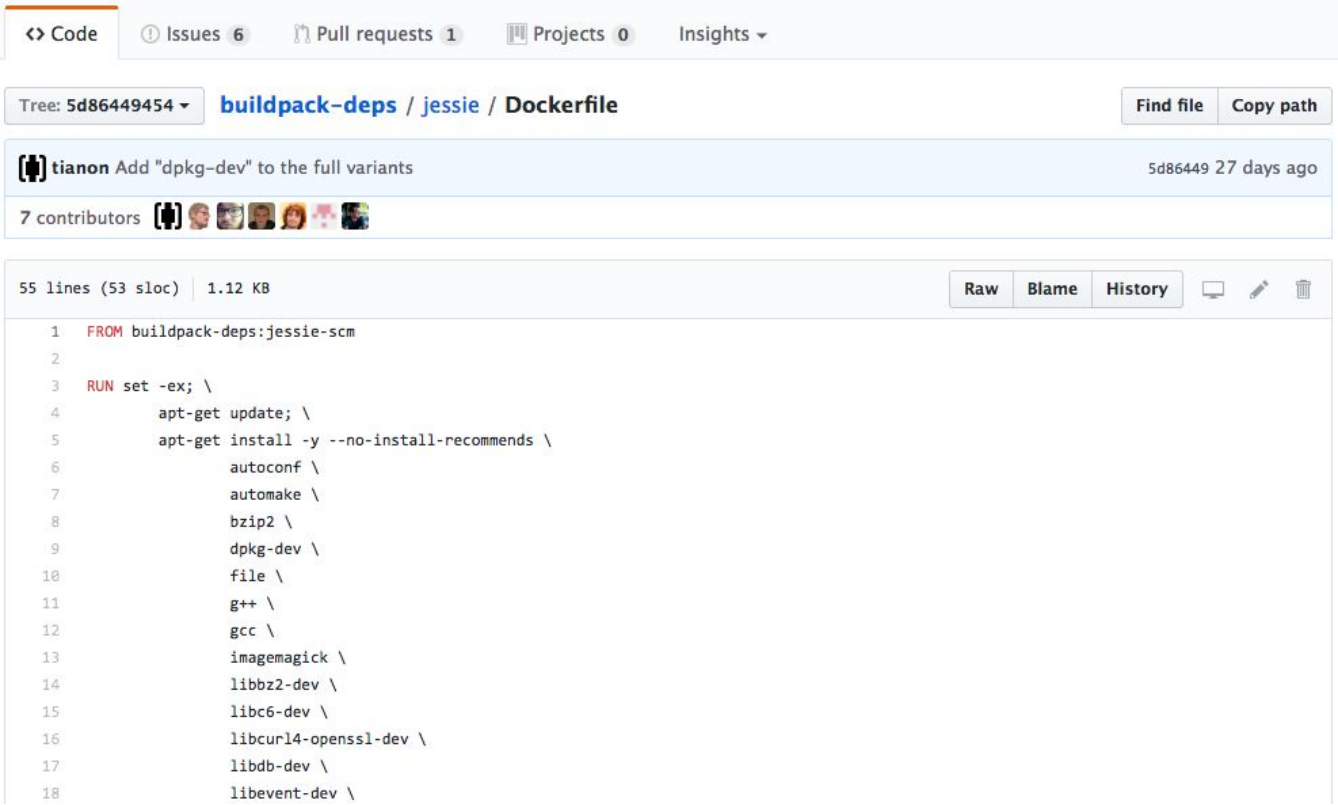
tianon Add more (future) helpful bits, TODO/comments, and exclude i386 for now — 3b33871 on Jun 16

1 contributor

125 lines (118 sloc) | 4.59 KB — Raw | Blame | History

```
1  FROM buildpack-deps:jessie
2
3  RUN set -ex; \
4          if ! command -v gpg > /dev/null; then \
5                  apt-get update; \
6                  apt-get install -y --no-install-recommends \
7                          gnupg2 \
8                          dirmngr \
9                  ; \
10                 rm -rf /var/lib/apt/lists/*; \
11         fi
12
13 # https://gcc.gnu.org/mirrors.html
14 ENV GPG_KEYS \
15 # 1024D/745C015A 1999-11-09 Gerald Pfeifer <gerald@pfeifer.com>
16         B215C1633BCA0477615F1B35A5B3A004745C015A \
17 # 1024D/B75C61B8 2003-04-10 Mark Mitchell <mark@codesourcery.com>
18         B3C42148A44E6983B3E4CC0793FA9B1AB75C61B8 \
```

What is the Gcc (5.4) container built from?



Code | Issues 6 | Pull requests 1 | Projects 0 | Insights

Tree: 5d86449454 ▾ **buildpack-deps** / jessie / **Dockerfile** — Find file | Copy path

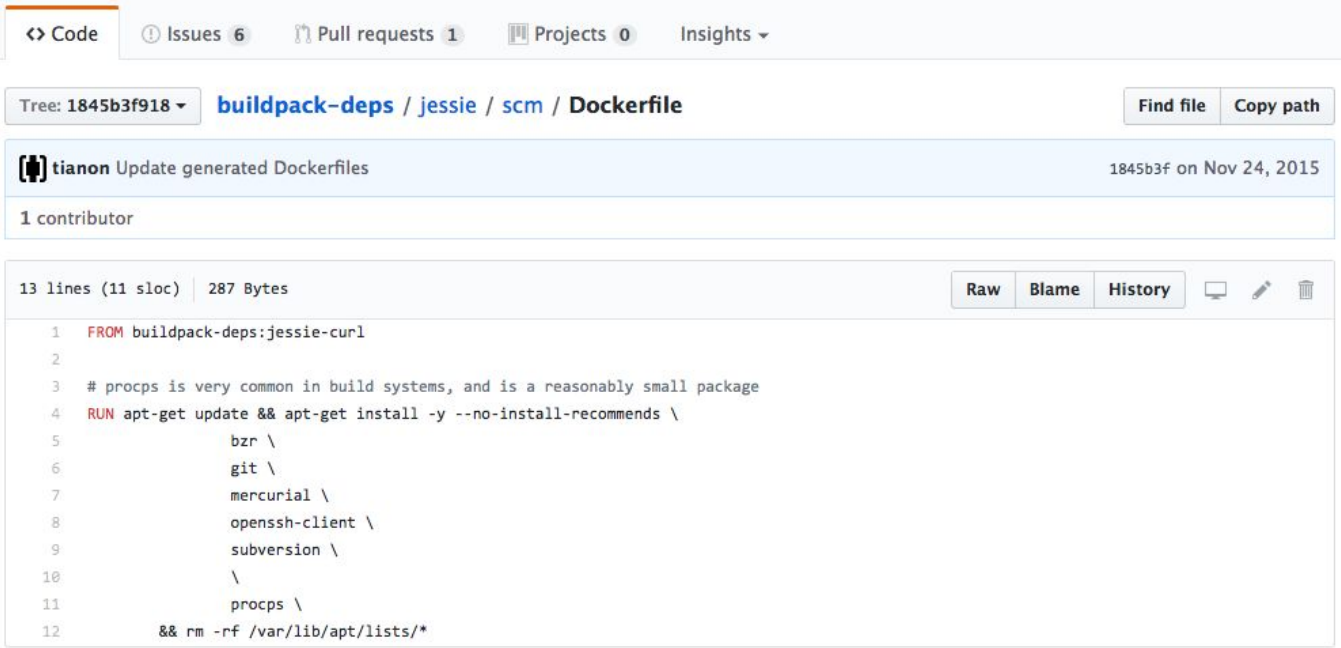
tianon Add "dpkg-dev" to the full variants — 5d86449 27 days ago

7 contributors

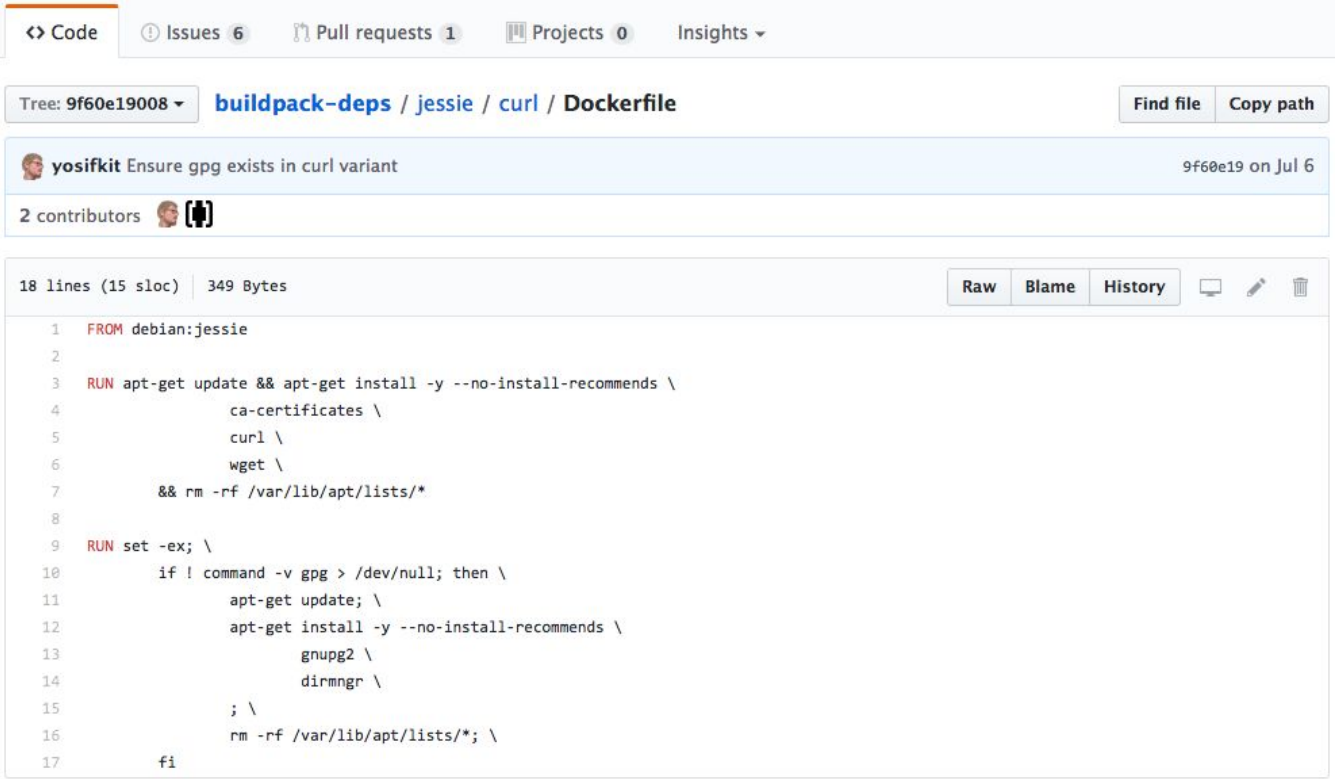
55 lines (53 sloc) | 1.12 KB — Raw | Blame | History

```
1  FROM buildpack-deps:jessie-scm
2
3  RUN set -ex; \
4          apt-get update; \
5          apt-get install -y --no-install-recommends \
6                  autoconf \
7                  automake \
8                  bzip2 \
9                  dpkg-dev \
10                 file \
11                 g++ \
12                 gcc \
13                 imagemagick \
14                 libbz2-dev \
15                 libc6-dev \
16                 libcurl4-openssl-dev \
17                 libdb-dev \
18                 libevent-dev \
```

What is the Gcc (5.4) container built from?



What is the Gcc (5.4) container built from?



Code | Issues 6 | Pull requests 1 | Projects 0 | Insights

Tree: 9f60e19008 ▾ **buildpack-deps** / jessie / curl / **Dockerfile** — Find file | Copy path

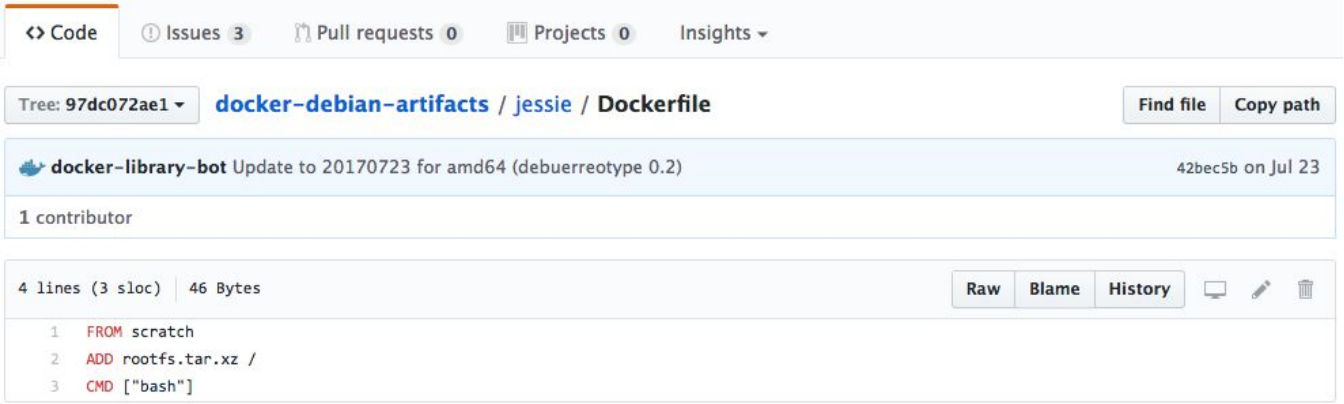
yosifkit Ensure gpg exists in curl variant — 9f60e19 on Jul 6

2 contributors

18 lines (15 sloc) | 349 Bytes — Raw | Blame | History

```
1  FROM debian:jessie
2
3  RUN apt-get update && apt-get install -y --no-install-recommends \
4                  ca-certificates \
5                  curl \
6                  wget \
7          && rm -rf /var/lib/apt/lists/*
8
9  RUN set -ex; \
10         if ! command -v gpg > /dev/null; then \
11                 apt-get update; \
12                 apt-get install -y --no-install-recommends \
13                         gnupg2 \
14                         dirmngr \
15                 ; \
16                 rm -rf /var/lib/apt/lists/*; \
17         fi
```

What is the Gcc (5.4) container built from?



Your first Docker container

We will now include and compile your test code directly from a Dockerfile

```
FROM gcc:5.4

# Add the test code
COPY test.c /opt

# Compile the test code
RUN gcc -v -o /opt/test.bin /opt/test.c
```

Your first Docker container

Let’s now build it with“test.c” and the Dockerfile files in a folder named “Test”:

```
$ docker build Test -t testcontainer
Sending build context to Docker daemon  10.24kB
Step 1/3 : FROM gcc:5.4
 ---> b87db7824271
Step 2/3 : COPY test.c /opt
 ---> f5478f7830ee
Step 3/3 : RUN gcc -v -o /opt/test.bin /opt/test.c
 ---> Running in c839379f1fbe
Using built-in specs.
COLLECT_GCC=gcc
[...]
Removing intermediate container c839379f1fbe
 ---> 2f0c6f89fdc0
Successfully built 2f0c6f89fdc0
Successfully tagged testcontainer:latest
```

Your first Docker container

..and we can run it:

```
$ docker run testcontainer /opt/test.bin
I just ran a very complex simulation and the result is 42
```

Your first Docker container

..and share it (old school):

```
$ docker save testcontainer > testcontainer.tar
```

```
$ docker load < testcontainer.tar
```

Your first Docker container

..and share it (Docker Hub):

```
$ docker tag testcontainer sarusso/testcontainer
$ docker push sarusso/testcontainer
The push refers to repository [docker.io/sarusso/testcontainer]
4e139ce93449: Pushed
8e5d12c6cc1e: Pushed
531d0aa62df3: Mounted from library/gcc
2ac9aba62fc1: Mounted from library/gcc
4e778218c153: Mounted from library/gcc
8f816dba9ff6: Mounted from library/gcc
7381522c58b0: Mounted from library/gcc
ecd70829ec3d: Mounted from library/gcc
d70ce8b0dad6: Mounted from library/gcc
18f9b4e2e1bc: Mounted from library/gcc
latest: digest: sha256:21563d1b6645af4cf73f01cc471b5f1a8bb902f7f1903bac4b9b878433eecf5e size: 2421
```

Versioning: hashes, tags, etc.

If we rebuild the testcontainer, the caching jumps in. It takes few seconds.

```
$ docker build Test -t testcontainer
Sending build context to Docker daemon  10.24kB
Step 1/3 : FROM gcc:5.4
 ---> b87db7824271
Step 2/3 : COPY test.c /opt
 ---> Using cache
 ---> f5478f7830ee
Step 3/3 : RUN gcc -v -o /opt/test.bin /opt/test.c
 ---> Using cache
 ---> 2f0c6f89fdc0
Successfully built 2f0c6f89fdc0
Successfully tagged testcontainer:latest
```

..this is possible thanks to *version hashes*

Versioning: hashes, tags, etc.

- An *hash* is the result of applying an hash function
- An hash function takes some input and generates a fixed-size output, like:

```
47e0b9046c241cc4653b876c2a8ab01341c00754
```

- A good hash function allows to virtually never have the same hash from different inputs.
- In both Git and Docker the input is your code, and and hash represents a unique (saved) state. Or, a particular point in your codebase “history”.
- Then, it happens that hashes can be linked together, forming hierarchies.
- A *tag* is a friendly name for an hash.

Versioning: hashes, tags, etc.

Let's have a look at the hashes for the first and second build

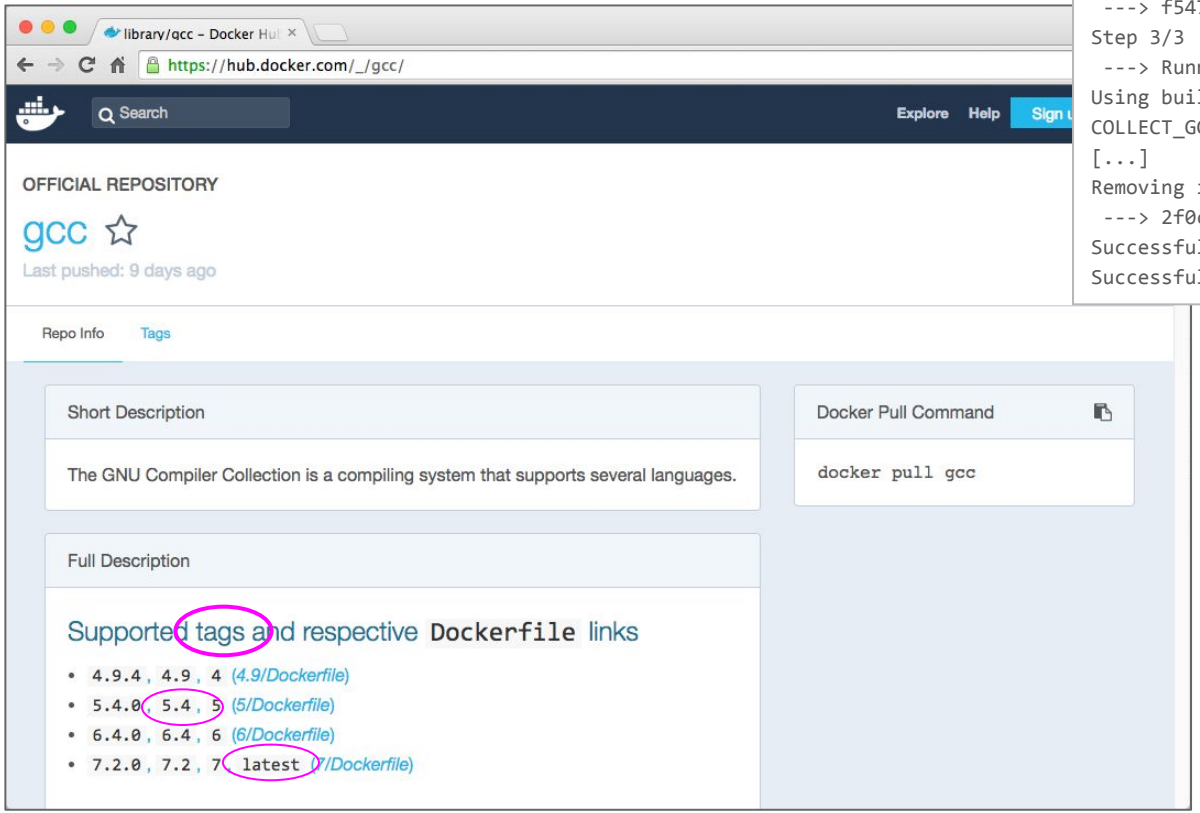
```
$ docker build Test -t testcontainer
Sending build context to Docker daemon  10.24kB
Step 1/3 : FROM gcc:5.4
 ---> b87db7824271
Step 2/3 : COPY test.c /opt
 ---> f5478f7830ee
Step 3/3 : RUN gcc -v -o /opt/test.bin /opt/test.c
 ---> Running in c839379f1fbe
Using built-in specs.
COLLECT_GCC=gcc
[...]
Removing intermediate container c839379f1fbe
 ---> 2f0c6f89fdc0
Successfully built 2f0c6f89fdc0
Successfully tagged testcontainer:latest
```

```
$ docker build Test -t testcontainer
Sending build context to Docker daemon  10.24kB
Step 1/3 : FROM gcc:5.4
 ---> b87db7824271
Step 2/3 : COPY test.c /opt
 ---> Using cache
 ---> f5478f7830ee
Step 3/3 : RUN gcc -v -o /opt/test.bin /opt/test.c
 ---> Using cache
 ---> 2f0c6f89fdc0
Successfully built 2f0c6f89fdc0
Successfully tagged testcontainer:latest
```

Versioning: hashes, tags, etc.

- Both Git and Docker implement versioning with hashes, which are fully deterministic, unlike version (incremental) numbers.
- In the Docker ecosystem everything is versioned
- For practical use, also the short hashes are allowed (and commonly used), which are the first 7 characters for Git (i.e. “`47e0b90`”) and the first 12 for Docker.
- If by chance two hashes in the system starts with the same short hash, you will be required to enter one more character or the full hash.

One step back




```
$ docker build Test -t testcontainer
Sending build context to Docker daemon  10.24kB
Step 1/3 : FROM gcc:5.4
 ---> b87db7824271
Step 2/3 : COPY test.c /opt
 ---> f5478f7830ee
Step 3/3 : RUN gcc -v -o /opt/test.bin /opt/test.c
 ---> Running in c839379f1fbe
Using built-in specs.
COLLECT_GCC=gcc
[...]
Removing intermediate container c839379f1fbe
 ---> 2f0c6f89fdc0
Successfully built 2f0c6f89fdc0
Successfully tagged testcontainer:latest
```

p.s. the tag “5.4” for the gcc Docker container is actually saying that the tag is “gcc:54”. Sorry for this! :(

The hash for the tag “gcc:5.4” tag is “b87db7824271”

Where do you save your code and Dockerfile?

Where do you save your code and Dockerfiles?

..on a versioning system.

Where do you save your code and Dockerfiles?

..on a versioning system.

There is no other alternative.

Do not work without versioning.

Seriously, don't.

→ Use Dropbox or Google Drive if you think that more professional versioning tools, like **Git**, are an overkill.

The importance of versioning with Docker

Docker allows to have everything up and running, including dependencies etc. with a single command.

This command trigger a build with a given set of dependencies (the ones you wrote to install in the Dockerfile)

Over time, you will probably make changes in your Dockerfiles and in your code.

If you use a versioning system, you can jump back in time to a particular version/hash, build it, and it will run exactly as it was running at that time

For managing multiple container versions simultaneously you can use tags

The importance of versioning with Docker

Docker allows to have everything up and running, including dependencies etc. with a single command.

This command trigger a build with a given set of dependencies (the ones you wrote to install in the Dockerfile)

Over time, you will probably make changes in your Dockerfiles and in your code.

If you use a versioning system, you can jump back in time to a particular **version**/**hash**, build it, and it will run exactly as it was running at that time

For managing multiple container versions simultaneously you can use **tags**

Recap on Docker

1) With Docker,your code will build and run in the exact same way, on every operating system, virtually forever.

2) If you want to give the code that generates the magic “42” answer to someone, they will just need two commands* to have everything up and running:

```
docker build or pull
docker run
```

**plus some arguments*

Personal take

- A versioning systems protects you first of all from yourself
- Using Docker with a versioning system allows to reach full reproducibility, starting from a repository name and a short hash for a point in time/version.
- Using them even for small personal/research projects helps a lot
- If someone gives you a code without version control *or* that requires dependencies:
  - First, put it under version control;
  - Second, create a Dockerfile with all the commands and dependencies you will need to set it up (which you will need anyway, by the way).
- ..and no, tomorrow you will not remember how you did. No one does. :)

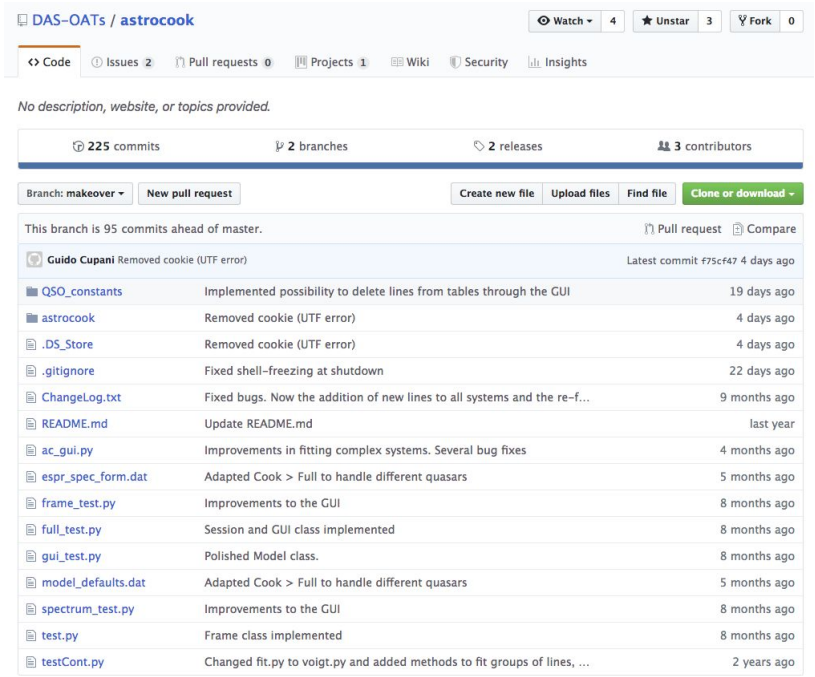
How about GUI programs?

Running a GUI program inside a container requires either a X11 server outside the container and forwarding the X11 socket, or other resorts like using VNC.

- Having a X11 server running outside a container and forward the X11 socket inside the container: ok on Linux, tricky on Mac, hard on Windows
- Better: embed a VNC server in the container and require just a VNC client
- Best: embed a WebVNC server in the container and require just a browser.

Practical example: Astrocook

A thousand ways to cook a spectrum



DAS-OATs / **astrocook**

Watch 4 | Unstar 3 | Fork 0

Code | Issues 2 | Pull requests 0 | Projects 1 | Wiki | Security | Insights

No description, website, or topics provided.

225 commits | 2 branches | 2 releases | 3 contributors

Branch: makeover | New pull request | Create new file | Upload files | Find file | Clone or download

This branch is 95 commits ahead of master. | Pull request | Compare

Guido Cupani Removed cookie (UTF error) — Latest commit f75cf47 4 days ago

QSO_constants	Implemented possibility to delete lines from tables through the GUI	19 days ago
astrocook	Removed cookie (UTF error)	4 days ago
.DS_Store	Removed cookie (UTF error)	4 days ago
.gitignore	Fixed shell-freezing at shutdown	22 days ago
ChangeLog.txt	Fixed bugs. Now the addition of new lines to all systems and the re-f…	9 months ago
README.md	Update README.md	last year
ac_gui.py	Improvements in fitting complex systems. Several bug fixes	4 months ago
espr_spec_form.dat	Adapted Cook > Full to handle different quasars	5 months ago
frame_test.py	Improvements to the GUI	8 months ago
full_test.py	Session and GUI class implemented	8 months ago
gui_test.py	Polished Model class.	8 months ago
model_defaults.dat	Adapted Cook > Full to handle different quasars	5 months ago
spectrum_test.py	Improvements to the GUI	8 months ago
test.py	Frame class implemented	8 months ago
testCont.py	Changed fit.py to voigt.py and added methods to fit groups of lines, …	2 years ago

README.md

Astrocook

A thousand ways to cook a spectrum!

DOI 10.5281/zenodo.1346737

Getting Started

To get a copy of Astrocook on your local machine:

```
git clone https://github.com/DAS-OATs/astrocook
```

Prerequisites

Astrocook requires the following packages to run:

- Astropy, including Specutils
- SciPy, and in particular NumPy and matplotlib
- LmFit
- StatsModels
- Cycler

Running the tests

The following tests are available:

- line_test.py: create a list of absoprtion lines from a spectrum and fit them with Voigt profiles.
- syst_test.py: create a list of CIV doublets from a spectrum and fit them with Voigt profiles.

To run the tests:

```
python <name-of-the-test>
```

Contributing

A CONTRIBUTING.md file will be soon uploaded to detail our code of conduct and the process for submitting pull requests to us.

Authors

- **Guido Cupani** – INAF-OATs
- **Giorgio Calderone** – INAF-OATs

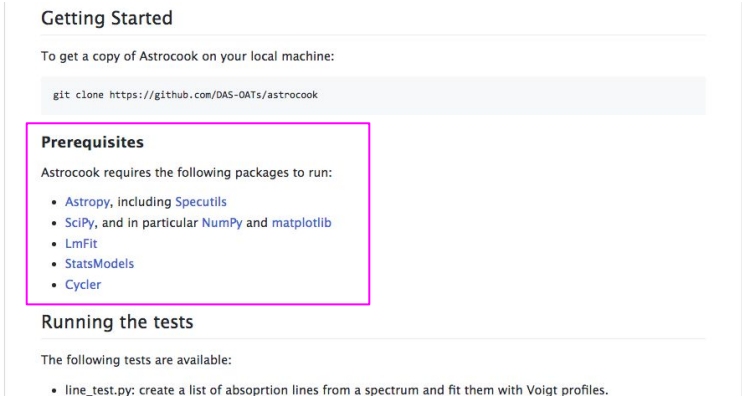
See also the list of contributors who participated in this project.

Practical example: Astrocook

A thousand ways to cook a spectrum

Typical Scientific Code that requires non-trivial dependencies.

It has a GUI, meaning that it requires even other “implicit” dependencies



Getting Started

To get a copy of Astrocook on your local machine:

```
git clone https://github.com/DAS-OATs/astrocook
```

Prerequisites

Astrocook requires the following packages to run:

- Astropy, including Specutils
- SciPy, and in particular NumPy and matplotlib
- LmFit
- StatsModels
- Cycler

Running the tests

The following tests are available:

- line_test.py: create a list of absoprtion lines from a spectrum and fit them with Voigt profiles.

Ok, let’s start to figure out how to install the dependencies.. on my 4 years old OS.

… or ...

Practical example: Astrocook

```
$ docker pull sarusso/astrocook
```

```
$ docker run -v$PWD:/data -p8590:8590 sarusso/astrocook
```

..and then open a browser on localhost:8590

Practical example: Astrocook

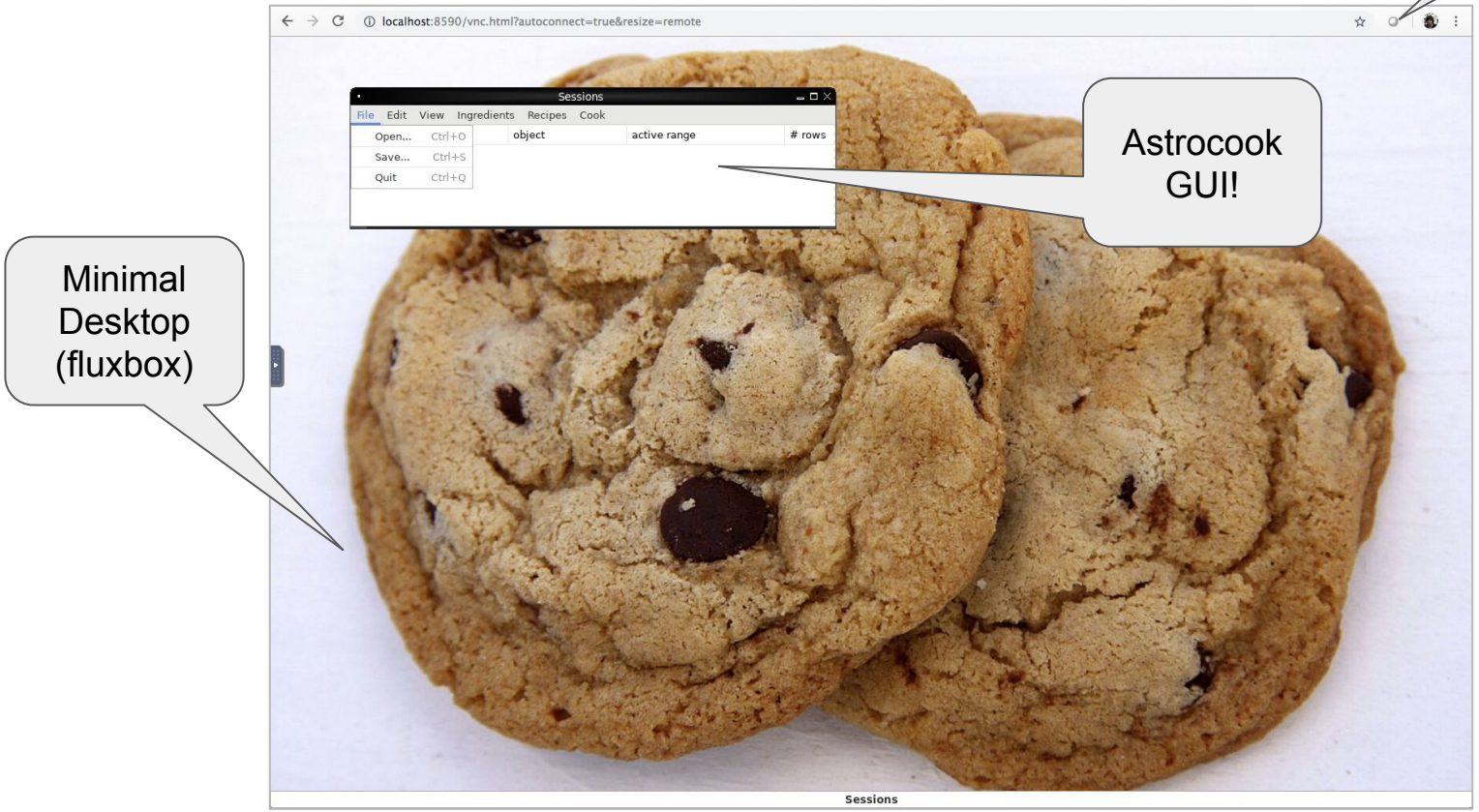
Clone the Astrocook repo, then:

```
$ docker build -f containers/Docker/Dockerfile . -t astrocook
```

```
$ docker run -v$PWD:/data -p8590:8590 astrocook
```

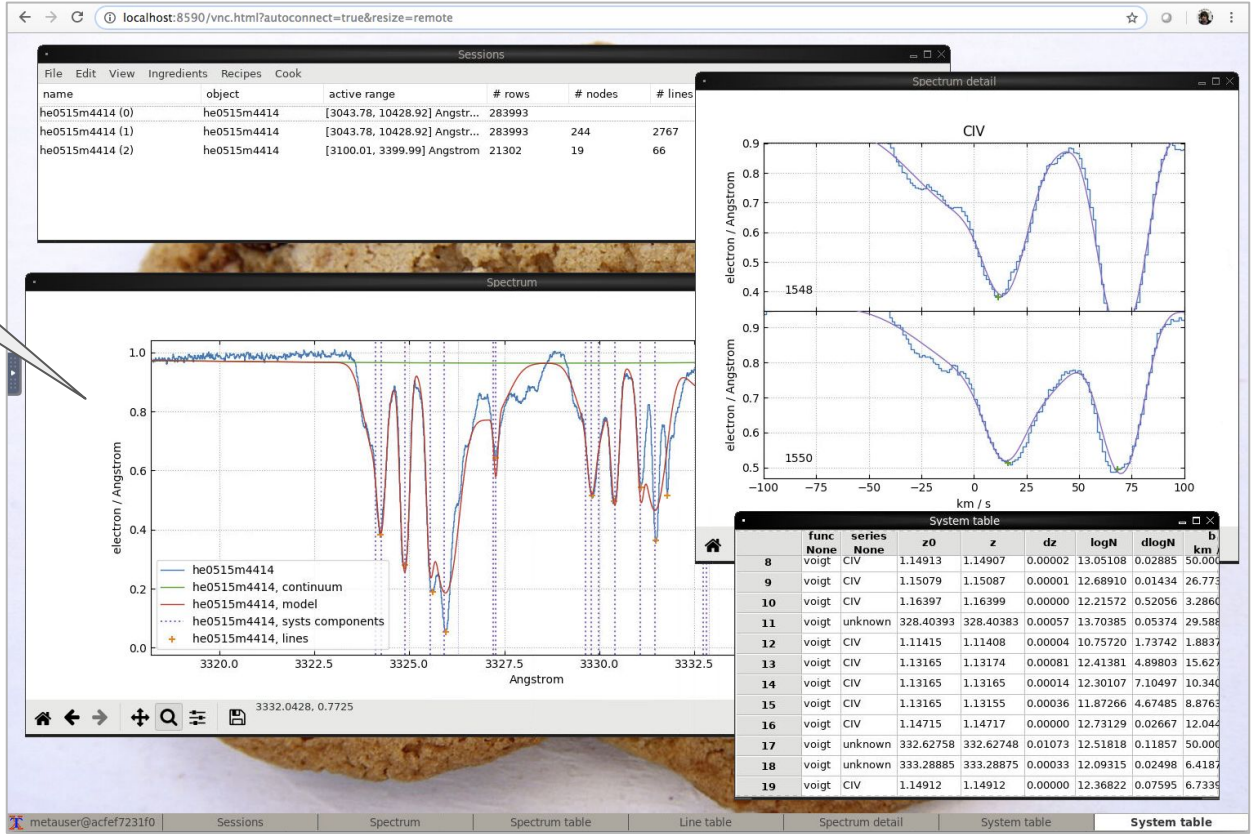
..and then open a browser on localhost:8590

Practical example: Astrocook



Practical example: Astrocook

Scientific stuff.



Hope it helps :)

Questions?

Stefano Alberto Russo

stefano.russo@inaf.it

Talk repository: https://github.com/sarusso/ModernSoftwareDevelopment