SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 16

Fabian Wermelinger
Harvard University
CS107 / AC207

Tuesday, October 25th 2022

LAST TIME

- Test-Driven development
- Testing code with `unittest` and `pytest`
- Computing and analyzing test coverage

TODAY

Main topics: ***Writing documentation, virtual environment and Docker containers***

Details:

- Documenting Python code (lecture 15)
- What are virtual environments and how to use them
- Configuration of Docker containers
- Using custom containers in CI

VIRTUAL PYTHON ENVIRONMENTS

Virtual environments in Python:

- Virtual environments are lightweight Python installations which are *isolated* from your base Python installation.
- They can be useful for many things:
  - Provide a minimal Python environment useful for all kinds of (automated) testing. ***We will test your project in a virtual environment***.
  - Avoid polluting your base Python installation (typically your system installation) with packages that do not belong there.
  - Avoid dependency conflicts. If your system installation has the most recent NumPy package installed but one of your software projects requires a *specific version* of NumPy.
  - Ensure reproducibility ***(important)***
  - Provide means to install packages on platforms where you do not have superuser privileges (and you do not want to install in your user site).

VIRTUAL PYTHON ENVIRONMENTS

There are different types of virtual environments:

- Python 3 ships with a `venv` module (https://docs.python.org/3/library/venv.html). A similar tool for Python 2 is `virtualenv` (https://pypi.org/project/virtualenv/).
  These are tools to create *lightweight* virtual environments and support only the Python version of your base installation.
- `pyenv` is a manager for different Python installations on your system and allows you to activate or deactivate a particular version or assign projects with specific Python versions. `pyenv` does not depend on a particular Python interpreter installed on your system (https://github.com/pyenv/pyenv). Packages are still installed via `pip`.
- Conda is an open source package management system and environment management system that runs on various operating systems (not lightweight). It automatically resolves dependencies of packages and allows you to create and switch to virtual environments. Conda does not use `pip` for package installation but can interface with it (https://docs.conda.io/en/latest/index.html).

VIRTUAL PYTHON ENVIRONMENTS

Lightweight virtual environments:

- Are based on an existing Python installation, called the *base installation*.
- Each virtual environment has *its own set of Python packages* installed in their `site` directories. ***Recall:*** `python -m site` to list your current site configuration (lecture 9).
- Such an environment is typically *isolated* from its base environment and contains only packages for the *core dependencies* (`pip` and `setuptools`). You could include packages from the base installation in a virtual environment however.
- Package management inside a virtual environment is enabled via `pip`.
- Your user site (see `python -m site --user-site`) *is considered part of the base installation* in a virtual environment.
- The Python standard library is *shared with the base installation*.

VIRTUAL PYTHON ENVIRONMENTS

Creating virtual environments:

- You can create a new virtual environment with the command:

```
1  $ python3 -m venv <path/to/new/venv>
```

This will simply create a new directory that contains the virtual environment. *If you want to remove the virtual environment later, simply remove the directory*.

- There are a few options available:

```
1  $ python3 -m venv --help
```

`--system-site-packages`
: Give the virtual environment access to the system `site-packages` directory.

`--symlinks/--copies`
: Try to use symbolic links/copies for executables.

`--without-pip`
: Skips installing or upgrading `pip` in the virtual environment.

`--upgrade-deps`
: Upgrade core dependencies (`pip/setuptools`) to latest version on the Python package index.

VIRTUAL PYTHON ENVIRONMENTS

What's inside a virtual environment?

- Virtual environments are *built on top of your base Python installation*. They are a lightweight version and contain only the *necessary directories* for a working Python environment:

  `pyvenv.cfg`
  : Configuration file for the virtual environment.

  `bin` (`Scripts` on Windows)
  : Subdirectory with Python executables (either symlinks or copies) and the *activation* and *deactivation* scripts.

  `lib/pythonX.Y/site-packages`
  : Subdirectory where the Python packages will be installed in the virtual environment. For a minimal setup this will either be empty or contains the core dependencies `pip` and `setuptools` only.

- *Note:* use of symbolic links is not recommended for Windows because double-clicking on a symbolic link in Windows will resolve eagerly and therefore bypass the virtual environment.

VIRTUAL PYTHON ENVIRONMENTS

Activating a virtual environment:

- To ***activate*** a virtual environment you must *source* the `bin/activate` script (for Bash and `zsh` shells)

```
$ source my_venv/bin/activate
```

- This will modify your `PATH` environment variable in the current shell to point to the Python interpreter in the virtual environment.
- When a virtual environment is used the `sys.prefix` and `sys.exec_prefix` point to the directories of the virtual environment and `sys.base_prefix` and `sys.exec_base_prefix` point to those of the base installation.
- You can determine if you are in a virtual environment using

```python
import sys
assert not (sys.prefix == sys.base_prefix)
```

VIRTUAL PYTHON ENVIRONMENTS

Summary:

- Virtual Python environments (`venv`) are *lightweight Python environments* that are useful for testing in isolation.
- You can use them to *ensure your software packages install as expected on a customers site* by not relying on your local Python environment.
- They are not activated by default in your shell (*unlike Conda that is modifying your `.bashrc`*). You are *activating and deactivating these environments explicitly*.
- When the virtual environment is no longer needed, you can simply remove the directory where it is defined.

WHAT IS A LINUX CONTAINER

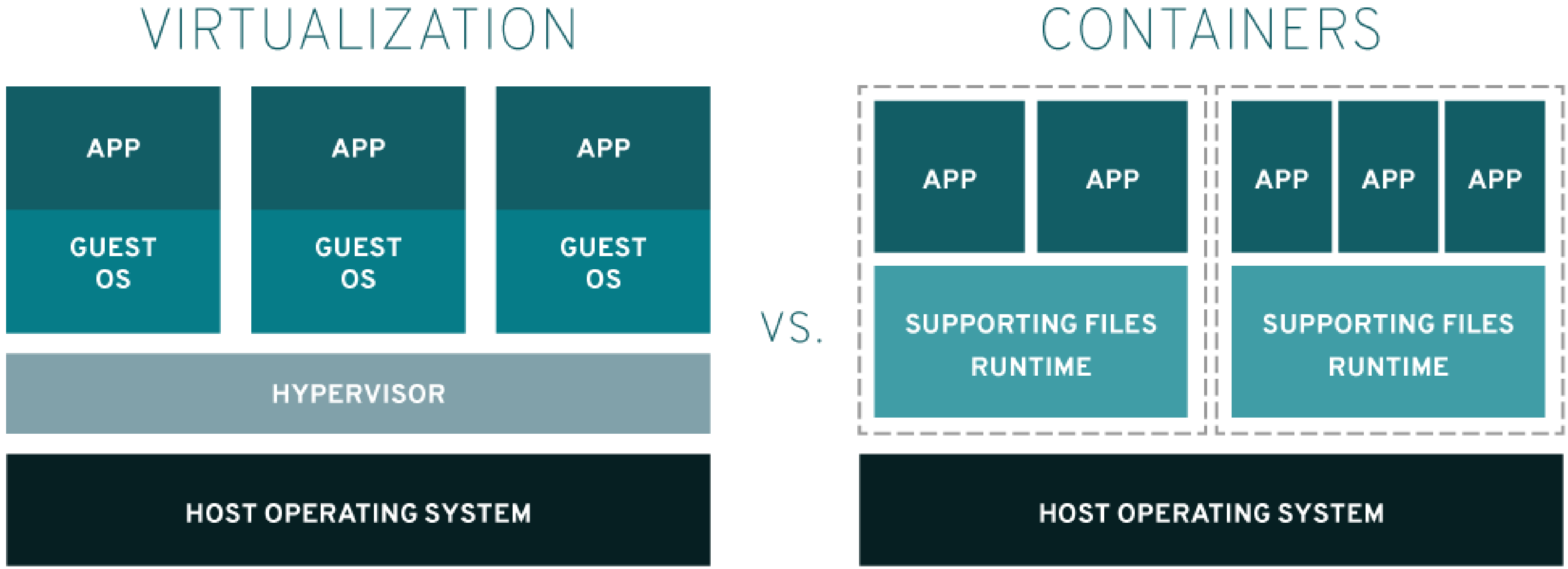
- A Linux container is a *set of one or more* processes that are *isolated* form the rest of the system. Running the Python interpreter is an example of a *process*. (***Recall:*** job/process management of lecture 3.)
- The concept of a container is similar to a virtual Python environment but it generalizes to *isolating* system processes, which is more powerful!
- All files necessary to run a container are provided within the container such that containers are ***portable*** among different platforms and are ***consistent*** for development, testing or production environments.
- Containers provide *great flexibility* and are easy to use, not only for CI. It is one of the reasons why they are so popular.

WHY WOULD YOU WANT TO USE CONTAINERS?

- Assume you develop an application that depends on a specific system configuration (libraries, services, etc.) that your company is using, e.g. on their servers.
- Replicating such a system environment on each developers laptop is not efficient as the configuration may change and requires individual maintenance.
- The solution to this problem is to pack this environment into a *container* and make it accessible to your developers.
- The container contains all the dependencies, libraries and other necessary files such that your application can execute in either development, testing or production environments.
- A container is *almost* like a Linux operating system on its own.

IS A LINUX CONTAINER A VIRTUAL MACHINE?

- Linux Containers ***are not*** virtual machines (e.g. VMware or VirtualBox)!
- Virtualization allows to run multiple operating systems side-by-side on a single hardware system.
- Containers *share the same operating system kernel* and *isolate* the application processes from the rest of the system.



(DOCKER) CONTAINERS

- Docker started in 2008 as a project of dotCloud with their container technology.
- There are three major components emerging from that technology required to ensure interoperability:
  1. ***Images*** (snapshot of container configuration, *not writeable*)
  2. ***Containers*** (component in which isolated processes run)
  3. ***Runtime specifications*** (protocol to communicate with host system)
- These components are more formally specified in the Open Container Initiative (OCI) to allow for an industry standard around container formats and runtimes → *Docker is not the only tool that can build images and run containers, see for example podman*.

OCI CONTAINERS

- Image based containers (OCI) are lightweight and many containers can run *with minimal overhead* and *in parallel* (***Recall:*** CI jobs run in parallel).
- Containers are designed to be *disposable*. Whenever you start the container anew, you start from the image snapshot defined in the container.
- → only the top-most layer of a container is *writable* and you could *commit* changes you make in this layer as a new layer to the container.
- Committing new layers is similar to Git. There is a `docker history <image ID>` command that lets you inspect the history of added layers.

OCI CONTAINERS

Container

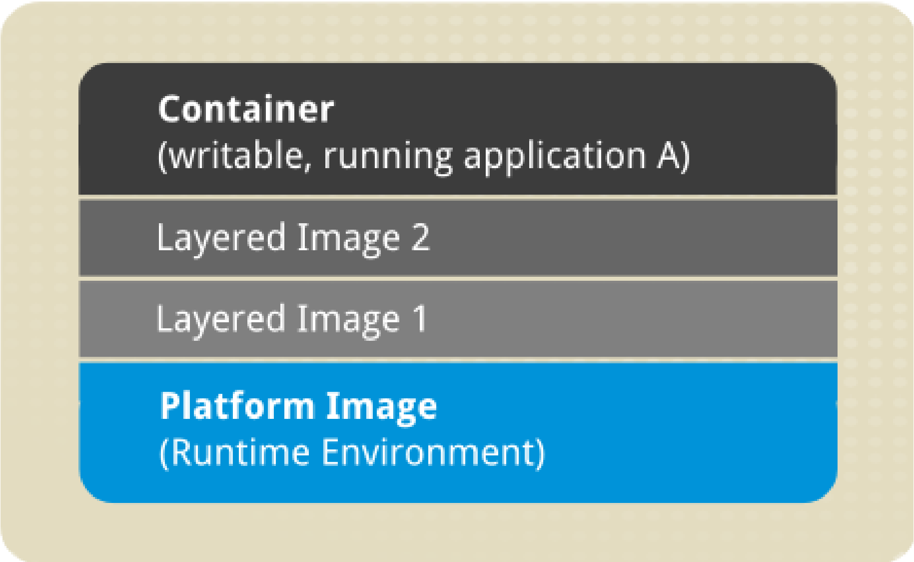
An active component in which an application (process) runs. Each container is based on an *image* that holds necessary configuration data. When you launch a container from an image, a *writable layer* is added on top of this image and typically disposed when the container terminates.

Layered image

A *static snapshot* of the containers configuration. Layered images are a *read-only* layers that are never modified, all changes are made in a top-most writable layer and *can be saved only by creating a new image*. Each image depends on one or more parent images.

Platform image

An image that has no parent. Platform images define the runtime environment, packages and utilities necessary for a containerized application to run (e.g. Ubuntu or other Linux distribution).



BUILDING AN IMAGE

Example: build an image for our Python test project and use it with CI

- Image definitions are written in a `Dockerfile` configuration file:

```
# base image
FROM docker.io/fedora:latest

RUN dnf -y update

# install basic Python development and frequently used packages
RUN dnf -y install \
    python-devel \
    python-build \
    python-numpy \
    python-pandas \
    python-matplotlib \
    python-pytest \
    python-pytest-cov

# we could add other custom Python code if we needed to.  For example the
```

- You can build a test image with `docker build -t cs107_lecture16 .` (assuming the `Dockerfile` is located in the *current directory*).

USING CUSTOM CONTAINERS IN CI

- The newly built image can be pushed to https://hub.docker.com (similar to GitHub) and make it accessible to the public.
- ***Never add SSH keys into an image!***
- Working with Docker is similar to working with Git:
  - You can run `docker pull` and `docker push` to pull and push images from/to a *container registry*
  - There are may such registries like https://hub.docker.com or https://quay.io for example. Their purpose is similar to what GitHub is for Git for example.
  - You can `docker commit` new layers (image) on top of other earlier layers. Whenever you change the file system state you would need to commit these changes to make them persistent.
  - You can inspect the history (layered images) in a container using `docker history`.

USING CUSTOM CONTAINERS IN CI

- We could now use this custom image and run it as a container in our CI jobs:

```yaml
jobs:
  test_coverage:

    # The type of runner that the job will run on (still required)
    runs-on: ubuntu-latest

    # here we specify the container image that we have built ourselves.  The
    # image already contains the Python environment for our needs and we can
    # skip to install a Python environment and dependencies below.  Procedure
    # is similar for other CI providers.
    # https://hub.docker.com/r/iacs/cs107_lecture16/tags
    container:
      image: iacs/cs107_lecture16:latest

    steps:
      - uses: actions/checkout@v3
```

- `Dockerfile` reference: https://docs.docker.com/engine/reference/builder

RECAP

- Documenting Python code
- What are virtual environments and how to use them
- Configuration of Docker containers
- Using custom containers in CI

Further reading:

- Virtual environments in Python (PEP405): https://peps.python.org/pep-0405/
- Python `venv` module: https://docs.python.org/3/library/venv.html
- Installing packages using `pip` and virtual environments: https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/#creating-a-virtual-environment
- What is a Linux container? https://www.redhat.com/en/topics/containers/whats-a-linux-container
- Introduction to Linux containers (RedHat)
- Getting started with containers (RedHat)
- Working with the `docker` command (RedHat)
- `Dockerfile` reference: https://docs.docker.com/engine/reference/builder