Big Data in Economics

Lecture 12: Docker

Grant McDermott
University of Oregon | EC 510

Table of contents

Prologue

Install Docker

- Linux
- Mac
- Windows (install varies by version)
    - Windows 10 Pro / Education / Enterprise
    - Windows 10 Home
    - Windows 7 / 8

Note: These lecture slides are mostly intended as a companion to the Rocker Wiki and the ROpenSci Docker tutorial. Use them as a quick-start reference for common Docker commands.

Docker 101

What is a container?

Have you ever...

- tried to install a program or run someone else's code, only to be confronted by a bunch of error messages?
- shared your code and data with someone else, only for them to be confronted by error messages?
- re-run your own analyses after a major package update, only to find that the code no longer works or the results have changed?

Containers are way to solve these (and many other) problems.

Docker

By far the most widely used and best supported container technology.

- Certainly true for the types of problems that we are concerned with in this course.
- So, while there are other container platforms around, when I talk about "containers" in this lecture, I'm really talking about Docker.

Why do we care?

I've already prompted some of the main reasons on the previous slide. But to sum things up with two ideas:

1. Reproducibility

If we can bundle our code and software in a Docker container, then we don't have to worry about it not working on someone else's system (and vice versa). Similarly, we don't have to worry about it not working on our own systems in the future (e.g. after package or program updates).

2. Deployment

There are many deployment scenarios (packaging, testing, etc.). Of particular interest to this course are big data pipelines where you want to deploy software quickly and reliably. Need to run some time-consuming code up on the cloud? Save time and installation headaches by running it through a suitable container, which can easily be deployed to a cluster of machines too.

The analogy

You know those big shipping containers used to transport physical goods?

They provide a standard format that can accommodate all manner of goods (TVs, fresh produce, whatever). Not only that, but they are stackable and can easily be switched between different modes of transport (ship, road, rail).

Docker containers are the software equivalent.

- physical goods <-> software
- transport modes <-> operating systems
- etc.

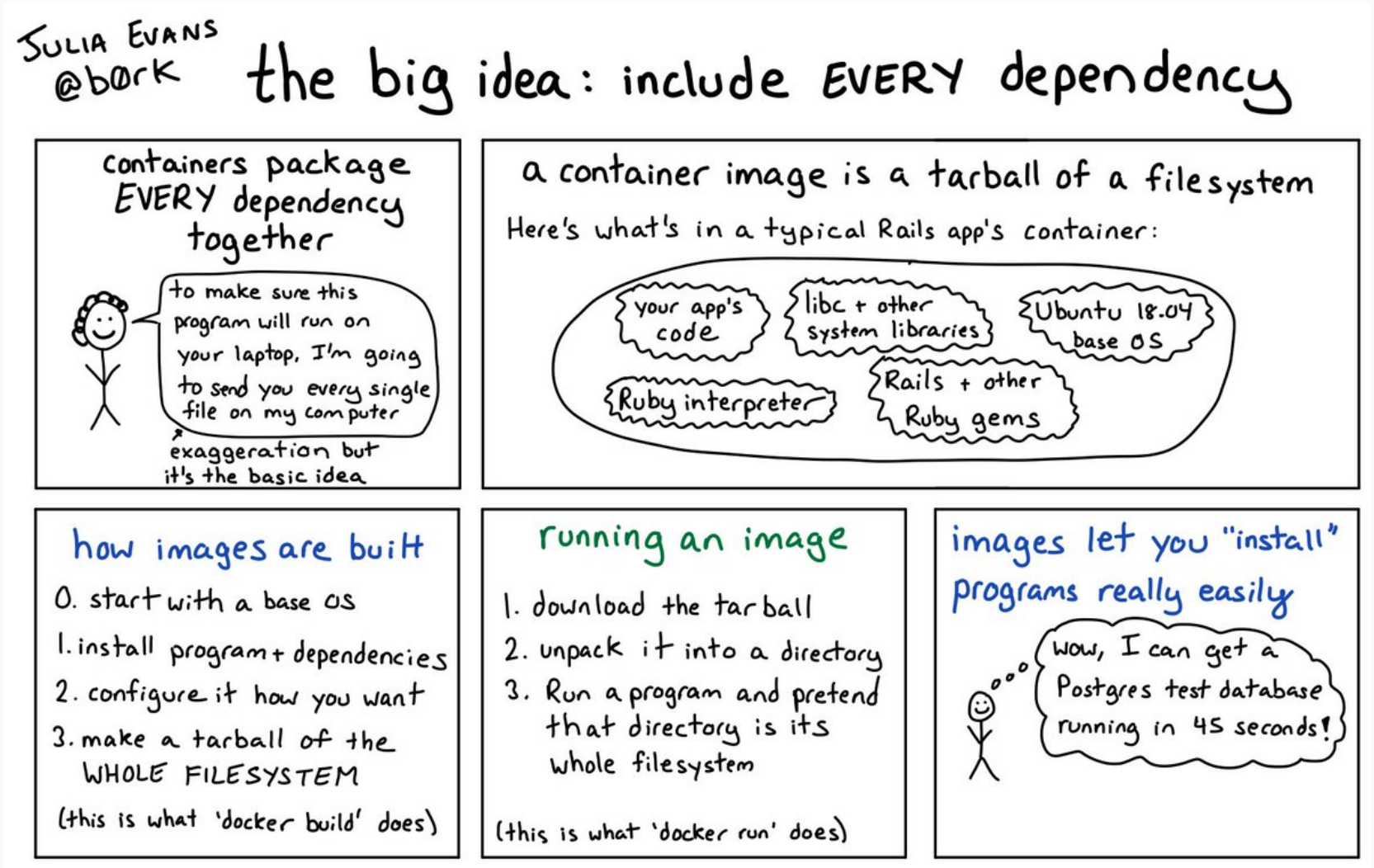
How it works

1. Start off with a stripped-down version of an operating system. Usually a Linux distro like Ubuntu.
2. Install *all* of the programs and dependencies that are needed to run your code.
3. (Add any extra configurations you want.)
4. Package everything up as a tarball.*

Summary: Containers are like mini, portable operating systems that contain everything needed to run some piece of software (but nothing more!).

* A format for storing a bunch of files as a single object. Can also be compressed to save space.

The big idea


JULIA EVANS
@b0rk
the big idea: include EVERY dependency
containers package EVERY dependency together
to make sure this program will run on your laptop, I'm going to send you every single file on my computer
exaggeration but it's the basic idea
a container image is a tarball of a filesystem
Here's what's in a typical Rails app's container:
your app's code
libc + other system libraries
Ubuntu 18.04 base OS
Ruby interpreter
Rails + other Ruby gems
how images are built
0. start with a base OS
1. install program + dependencies
2. configure it how you want
3. make a tarball of the WHOLE FILESYSTEM
(this is what 'docker build' does)
running an image
1. download the tarball
2. unpack it into a directory
3. Run a program and pretend that directory is its whole filesystem
(this is what 'docker run' does)
images let you "install" programs really easily
wow, I can get a Postgres test database running in 45 seconds!

Quick terminology clarification

Image ~ This is (basically) the tarball that we talked about on the previous two slides. It's a set of layers and instructions for building a container.

Container ~ A container is a running instance of an image. You can have many containers (i.e. running instances) of the same image.

Analogy: Think of the image as the recipe and the container as the cake. You can use the recipe to make many versions of the same cake (even if you only ever plan to make one). And you can share the recipe with others to make their own cake.

Rocker = R + Docker

It should now be clear that Docker is targeted at (and used by) a bewildering array of software applications.

In the realm of economics and data science, that includes every major open-source programming language and software stack.[1] For example, you could download and run a Julia container right now if you so wished.

But for this course, we are primarily concerned with Docker images that bundle R applications.

The good news is that R has outstanding Docker support, primarily as a result of the **Rocker Project** (website / GitHub).

- For the rest of today's lecture we will be using images from Rocker (or derivatives).

[1] It's *possible* to build a Docker image on top of proprietary software (example). But license restrictions and other limitations make this complicated. I hardly ever see it done in the wild.

Examples

Base R container

For our first example, let's fire up a simple container that contains little more than a base R installation.

```
$ docker run --rm -ti rocker/r-base
```

This will take a little while to download the first time. But afterwards the container will be ready and waiting for immediate deployment ("instantiation") on your system.

A quick note on these `docker run` flags:

- `--rm` Automatically remove the container once it exits (i.e. clean up).
- `-ti` Launch with interactive (`i`) shell/terminal (`t`).
- Type `man docker run` to see a full list of flag options.

To see a list of running containers on your system, in a new terminal window type:

```
$ docker ps
```

Base R container (cont.)

Your base R container should have launched directly into R. To exit the container, simply quit R.

```
R> q()
```

Check that it worked:

```
$ docker ps
```

BTW, if you don't want to launch directly into your container's R console, you can instead start it in the bash shell.

```
$ docker run --rm -ti rocker/r-base /bin/bash
```

This time to close and exit the container, you need to exit the shell, e.g.

```
root@09dda673a187:/# exit
```

RStudio+ container

The Rocker Project works by layering Docker images on top of each other in a grouped stack.

An important group here is the versioned stack that includes RStudio.

- For example, the "rstudio" image builds on top of the "r-ver" image (which itself is a versioned "r-base" image).

I'm going to collectively refer to images in this stack as **RStudio+**.

- It's not that each image in the stack contains exactly the same things...
- But they do share some important common features that will make the RStudio+ shorthand convenient for these lecture slides.

Everyone clear on what I mean by "RStudio+"? Good.

RStudio+ container (cont.)

Let's try the "tidyverse" image, which is basically base R + RStudio + tidyverse packages.

Again, this will take a minute or three to download and extract the first time. But it will be ready for immediate deployment in the future.

```
$ docker run -d -p 8787:8787 -e PASSWORD=mypassword rocker/tidyverse
```

If you run this... nothing seems to happen. Don't worry, I'll explain on the next slide.

But first, two quick asides on usernames/passwords:

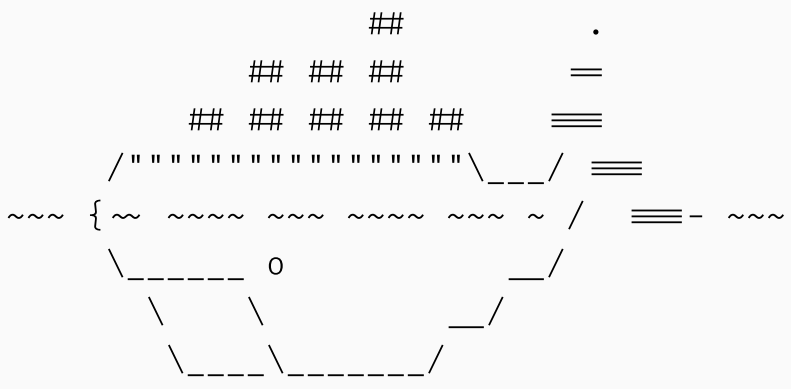
- All RStudio+ images in the Rocker stack require a password. Pretty much anything you want except "rstudio", which is the default username...
- If you don't like the default "rstudio" username, you can choose your own by adding `-e USER=myusername` to the above command (the extra `-e` flag is required).

RStudio+ container (cont.)

Unlike, the "r-base" container, this time we aren't immediately taken to our R environment.
Reason: Our container is running RStudio Server, which needs to be opened up in a browser.

So we need to point our browsers to the relevant IP address (plus the assigned `8787` port):

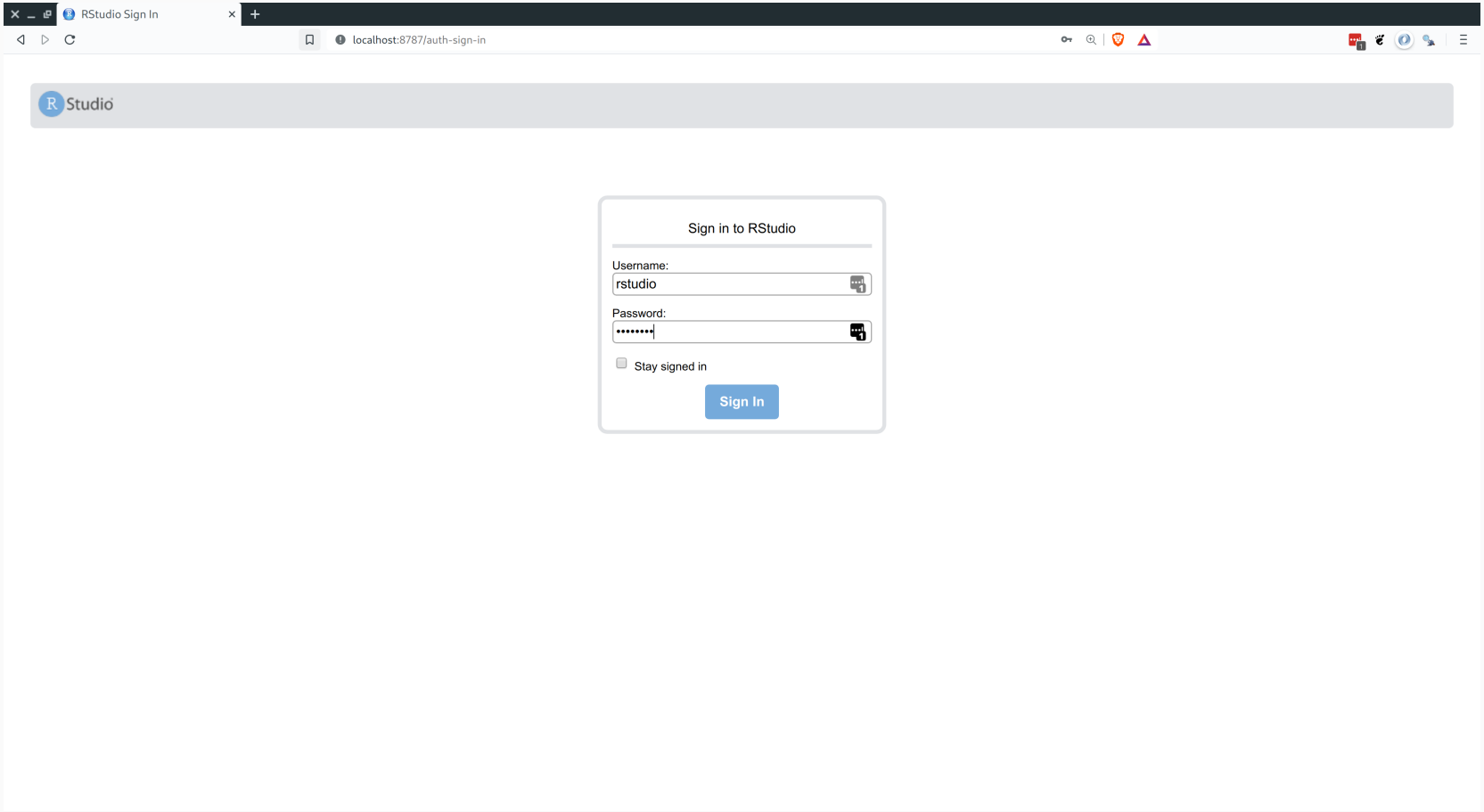
- **Linux:** `http://localhost:8787`
- **Mac/Windows:** Get your IP address by typing `$ docker inspect <containerid> | grep IPAddress` (see here). Alternatively, this IP address was also displayed when you first launched your Docker Quickstart Terminal. E.g. The below user would navigate their browser to `192.168.99.100:8787`.



```
docker is configured to use the default machine with IP 192.168.99.100
```

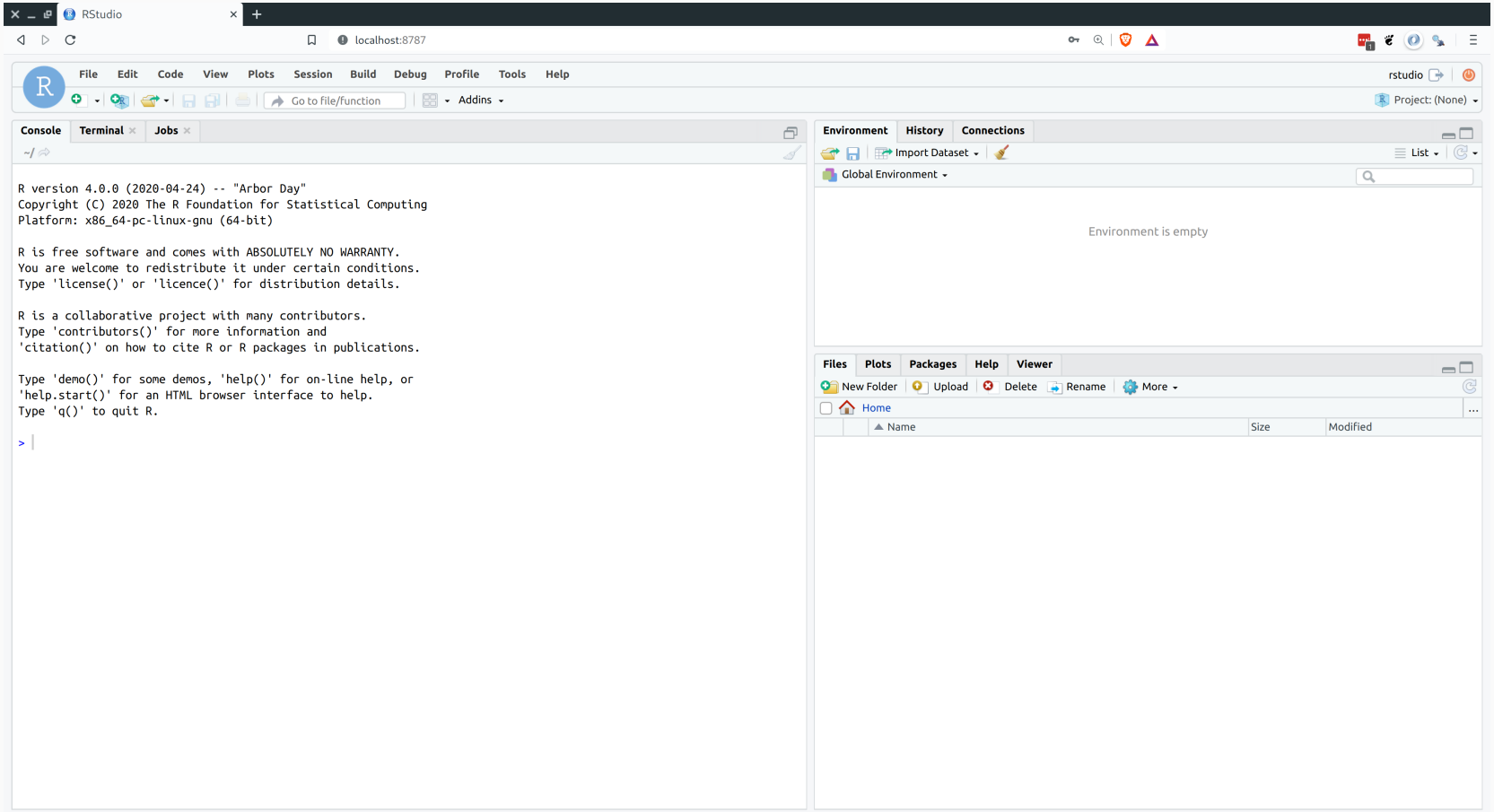
RStudio+ container (cont.)

Here's the login-in screen that I see when I navigate my browser to the relevant URL.



RStudio+ container (cont.)

And here I am in RStudio Server running through Docker! (Pro-tip: Hit F11 to go full-screen.)



RStudio+ container (cont.)

To stop this container, open up a new terminal window and grab the container ID with `$ docker ps` (it will be the first column). Then run:

```
$ docker stop <containerid>
```

Note: We actually ran this container as a background process, because we used the `-d` flag when we first instantiated it, i.e.

```
$ docker run -d -p 8787:8787 -e PASSWORD=mypassword rocker/tidyverse
```

If you dropped the `-d` flag and re-ran the above command, your terminal would stay open as an ongoing process. (Try this yourself.)

- Everything else would remain the same. You'd still navigate to `<IPADDRESS>:8787` to log in, etc.
- However, I wanted to mention this non-background process version because it offers another way to shut down the container: Simply type `CTRL+c` in the (same, ongoing process) Terminal window. Again, try this yourself.
- Confirm that the container is stopped by running `$ docker ps`.

Sharing files with a container

Share files by mounting volumes

Each container runs in a sandboxed environment and cannot access other files and directories on your computer unless you give it explicit permission.

To share files with a container, the `-v` (mount volume) flag is your friend.

- Adopts a `LHS:RHS` convention, where `LHS` = `path/on/your/computer/` and `RHS` = `path/on/the/container`.

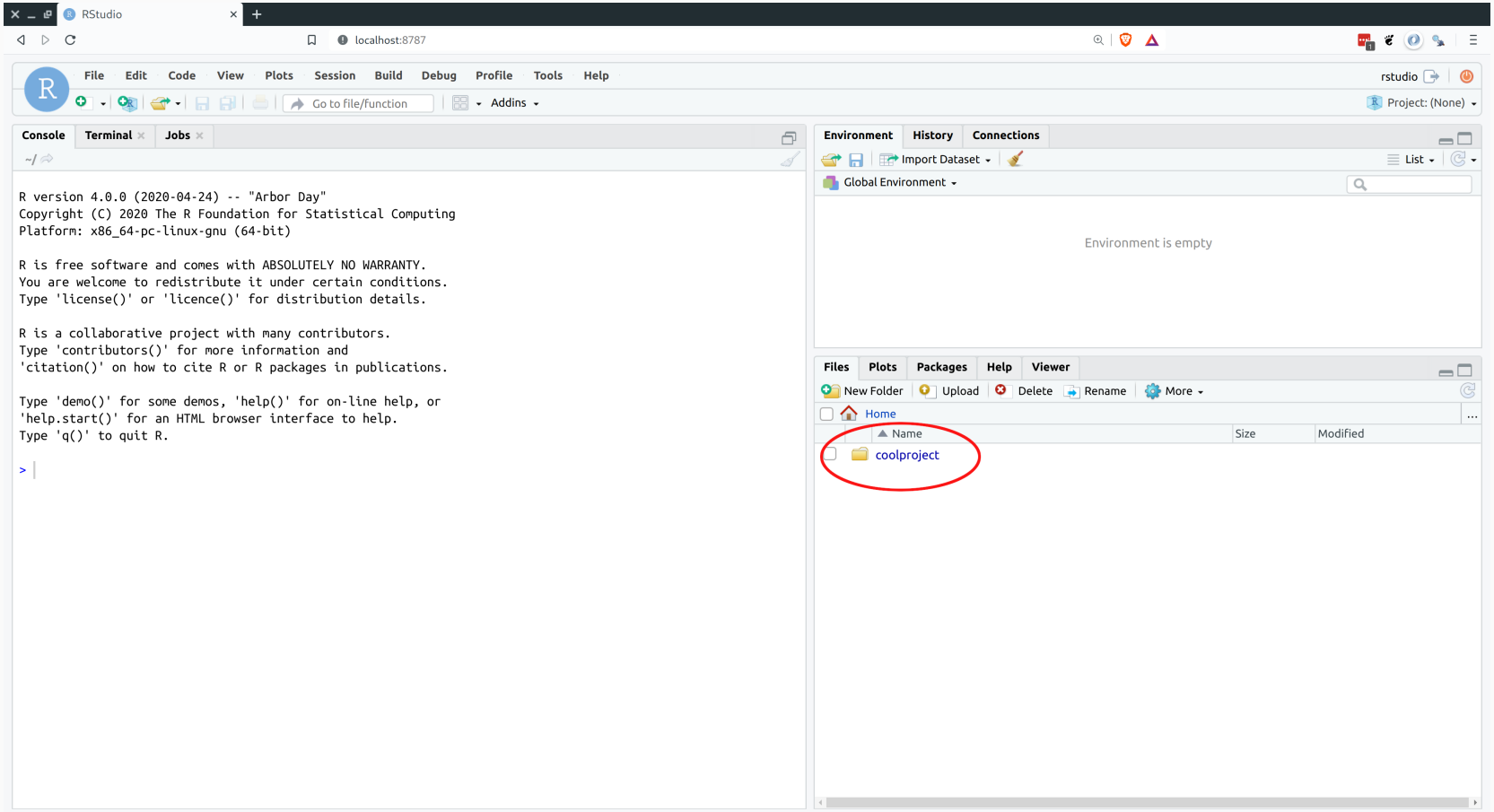
For example, say I have a folder on my computer located at `/home/grant/coolproject`. I can make this available to my "tidyverse" container by running:

```
$ docker run -v /home/grant/coolproject:/home/rstudio/coolproject -d -p 8787:8787
```

PS — I'll get back to specifying the correct RHS path in a couple slides.

coolproject

The coolproject directory is now available from RStudio running on the container.



Choosing the RHS mount point

In the previous example, I specified the RHS mount point as `/home/rstudio/coolproject`.

How did I know this would work?

The short answer is that `/home/rstudio` is the default user's home directory for images in the RStudio+ stack. If you're running a container from this stack, you should almost always start your RHS with this path root.

- Exception: If you assigned a different default user than "rstudio" (back here).

OTOH, the `/coolproject` directory name is entirely optional. You could call it anything you want. Giving it the same name as the directory on your computer obviously helps to avoid confusion, though.

- Similarly, nothing is stopping you from adding a couple of parent directories. I could have used `-v /home/grant/coolproject:/home/rstudio/parentdir/coolproject` and it would have worked fine.

Choosing the RHS mount point (cont.)

So, the RHS mount point for RStudio+ containers is (almost) always `/home/rstudio`.

What about other containers?

- E.g. For r-base containers there's no "rstudio" user so the above definitely won't work. (Fun fact: When you run an r-base container you are actually logged in as root.)

In truth, the specific RHS mount point is less important for non-RStudio+ containers.[1] Still, I recommend a general strategy of mounting external volumes on the dedicated `/mnt` directory that is standard on Linux.[2] For example:

```
$ docker run -it --rm -v /home/grant/coolproject:/mnt/coolproject r-base
```

[1] It only really matters for the first group because RStudio Server limits how and where users can access files. This is a security feature we'll revisit in the next lecture on cloud computing.

[2] Remember, Docker images are basically just miniaturized, portable Linux OSs.

Next steps

Further reading

- Rocker Wiki
- ROpenSci Docker Tutorial

Next class: Cloud computing with Google Compute Engine