Data Science for Economists

Lecture 13: Docker

Grant McDermott
University of Oregon | EC 607

Table of contents

1. Prologue
2. Docker 101
3. Examples
    - Base R
    - RStudio+
4. Sharing files with a container
5. Next steps

Prologue

Install Docker

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

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?

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.

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 data science 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).

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.*

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

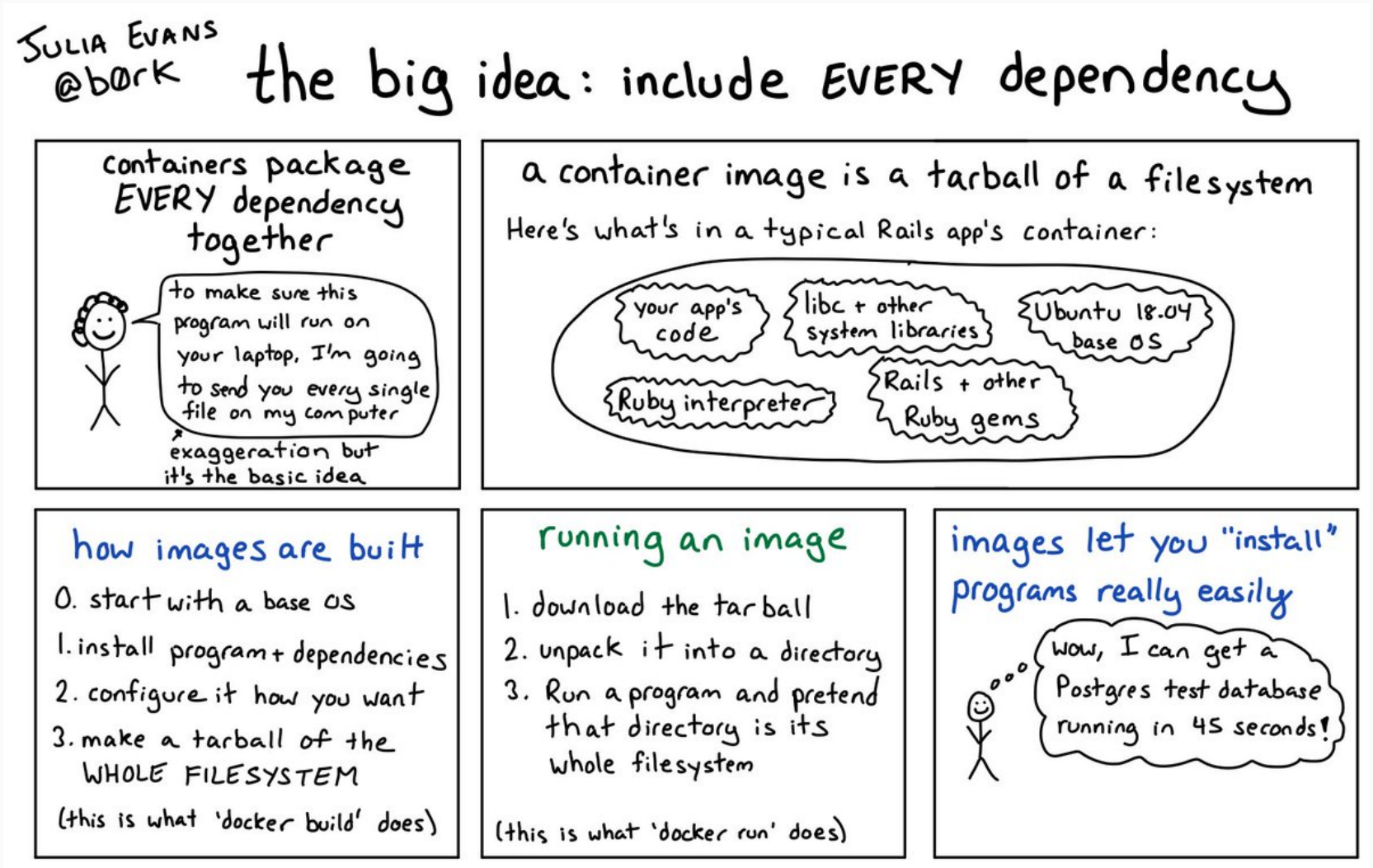
How it works

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 the tarball that we talked about on the previous two slides. It is 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.

Quick terminology clarification

Image ~ This is the tarball that we talked about on the previous two slides. It is 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.

[1] It's possible to build a Docker image on top of proprietary software (example). But license restrictions make this complicated. I've rarely ever see it done in practice.

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 thanks to 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 make this complicated. I've rarely ever see it done in practice.

Examples

Example 1: Base R

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

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

This will take a little while to download the first time (GIF on next slide). But the container will be ready and waiting for immediate deployment on your system thereafter.

Example 1: Base R

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

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

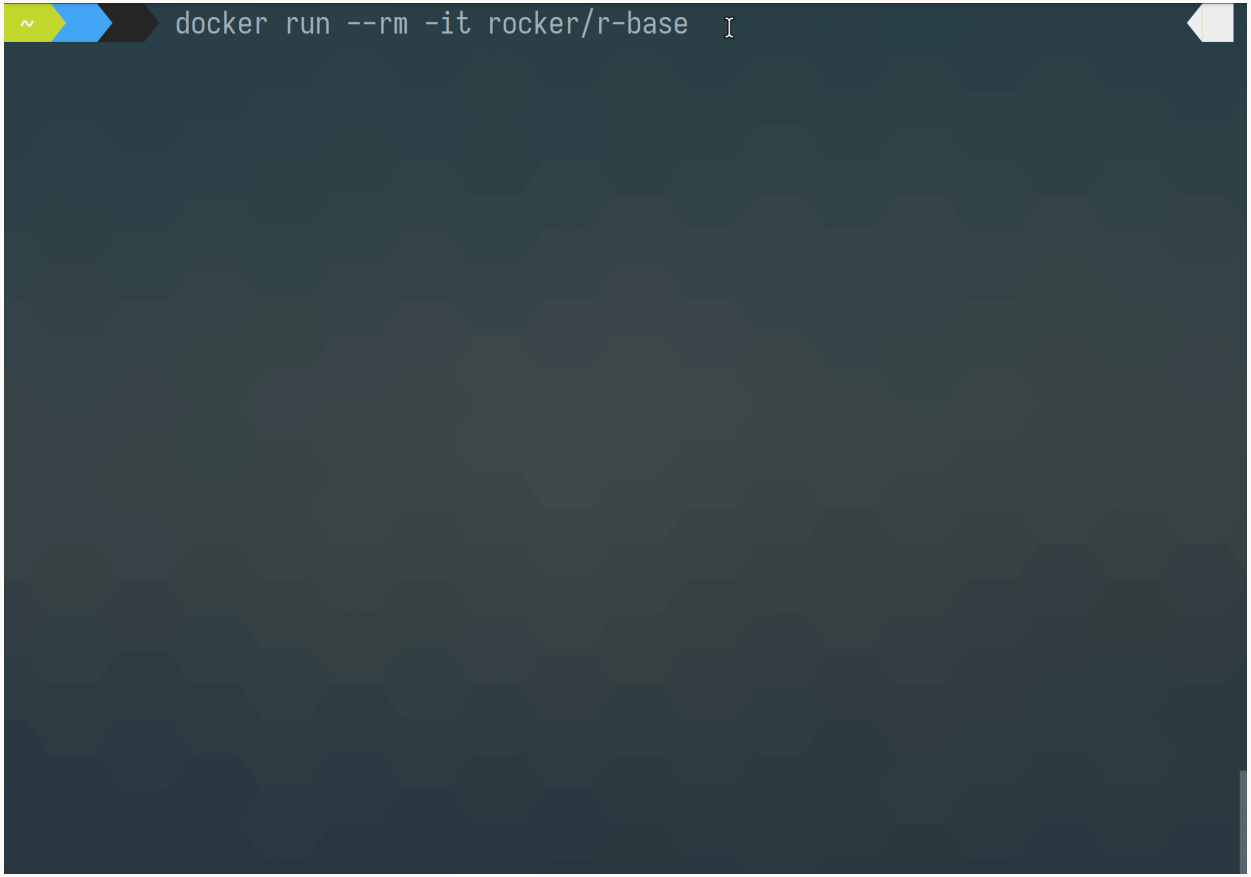
This will take a little while to download the first time (GIF on next slide). But the container will be ready and waiting for immediate deployment on your system thereafter.

A quick note on these `docker run` flags:

- `--rm` Automatically remove the container once it exits (i.e. clean up).
- `-it` Launch with interactive (`i`) shell/terminal (`t`).
- For a full list of flag options, see here.

Example 1: Base R (cont.)

As promised, here is a GIF of me running the command on my system. The whole thing takes about a minute and takes me directly into an R session.



Example 1: Base R (cont.)

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

```
$ docker ps
```

You should see something like:

```
CONTAINER ID   IMAGE           COMMAND   CREATED          STATUS          PORTS
1fcdee074beb   rocker/r-base   "R"       35 seconds ago   Up 35 seconds
```

Example 1: Base R (cont.)

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

```
$ docker ps
```

You should see something like:

```
CONTAINER ID   IMAGE            COMMAND   CREATED          STATUS          PORTS
1fcdee074beb   rocker/r-base    "R"       35 seconds ago   Up 35 seconds
```

The container ID (here: "1fcdee074beb") is probably the most important bit of information.

- We'll be using container IDs later in the lecture. For now, just remember that you can grab them with the `$ docker ps` command.

Example 1: Base R (cont.)

Your base R container should have launched directly into R. Feel feel to kick the tyres. Run a regression on the mtcars dataset, do some addition, etc.

Example 1: Base R (cont.)

Your base R container should have launched directly into R. Feel feel to kick the tyres. Run a regression on the mtcars dataset, do some addition, etc.

To exit the container, simply quit R.

```
R> q()
```

Check that it worked:

```
$ docker ps
```

Example 1: Base R (cont.)

Your base R container should have launched directly into R. Feel feel to kick the tyres. Run a regression on the mtcars dataset, do some addition, etc.

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 -it rocker/r-base /bin/bash
```

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

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

Example 2: RStudio+

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.

image	description	size	pull
r-ver	Version-stable base R & src build tools	0B 16 layers	docker pulls 584k
rstudio	Adds rstudio	355.1MB 21 layers	docker pulls 5.7M
tidyverse	Adds tidyverse & devtools	0B 26 layers	docker pulls 2M
verse	Adds tex & publishing-related packages	1.2GB 30 layers	docker pulls 723k
geospatial	Adds geospatial packages on top of 'verse'	0B 32 layers	docker pulls 249k
shiny	Adds shiny server on top of 'r-ver'	0B 13 layers	docker pulls 938k
shiny-verse	Adds tidyverse packages on top of 'shiny'	0B 14 layers	docker pulls 177k
binder	Adds requirements to 'geospatial' to run repositories on mybinder.org	0B 45 layers	docker pulls 74k
ml	Adds python and Tensorflow to 'tidyverse'	0B 47 layers	docker pulls 29k
ml-verse	Adds python and Tensorflow to 'verse'	0B 47 layers	docker pulls 29k

Example 2: RStudio+

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.

image	description	size	pull
r-ver	Version-stable base R & src build tools	0B 16 layers	docker pulls 584k
rstudio	Adds rstudio	355.1MB 21 layers	docker pulls 5.7M
tidyverse	Adds tidyverse & devtools	0B 26 layers	docker pulls 2M
verse	Adds tex & publishing-related packages	1.2GB 30 layers	docker pulls 723k
geospatial	Adds geospatial packages on top of 'verse'	0B 32 layers	docker pulls 249k
shiny	Adds shiny server on top of 'r-ver'	0B 13 layers	docker pulls 938k
shiny-verse	Adds tidyverse packages on top of 'shiny'	0B 14 layers	docker pulls 177k
binder	Adds requirements to 'geospatial' to run repositories on mybinder.org	0B 45 layers	docker pulls 74k
ml	Adds python and Tensorflow to 'tidyverse'	0B 47 layers	docker pulls 29k
ml-verse	Adds python and Tensorflow to 'verse'	0B 47 layers	docker pulls 29k

Allows us to easily spin up different versions of R (3.6.1, 4.0.2, etc), plus extra layers.

Example 2: RStudio+ (cont.)

Let's try the `tidyverse` image from this versioned stack, which layers base R + RStudio + tidyverse. I'll specify R 4.0.0 as my base image.

Again, this next line will take a minute or three to download and extract the first time. But the container will be ready for immediate deployment on your system thereafter.

Example 2: RStudio+ (cont.)

Let's try the `tidyverse` image from this versioned stack, which layers base R + RStudio + tidyverse. I'll specify R 4.0.0 as my base image.

```
$ docker run -d -p 8787:8787 -e PASSWORD=mypassword rocker/tidyverse:4.0.0
```

- `-d` Detach (i.e. run as background process).
- `-p 8787:8787` Share a port with the host computer's browser.
- `-e PASSWORD=mypassword` Password for logging on to RStudio Server.
- `rocker/tidyverse:4.0.0` Use the `tidyverse` image built on top of R 4.0.0.

Example 2: RStudio+ (cont.)

Let's try the `tidyverse` image from this versioned stack, which layers base R + RStudio + tidyverse. I'll specify R 4.0.0 as my base image.

```
$ docker run -d -p 8787:8787 -e PASSWORD=mypassword rocker/tidyverse:4.0.0
```

- `-d` Detach (i.e. run as background process).
- `-p 8787:8787` Share a port with the host computer's browser.
- `-e PASSWORD=mypassword` Password for logging on to RStudio Server.
- `rocker/tidyverse:4.0.0` Use the `tidyverse` image built on top of R 4.0.0.

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

- Confirm for yourself that it's actually running with `$ docker ps`. (Mac and Windows users should definitely do this because you'll need the container ID shortly.)

Example 2: RStudio+ (cont.)

Let's try the `tidyverse` image from this versioned stack, which layers base R + RStudio + tidyverse. I'll specify R 4.0.0 as my base image.

```
$ docker run -d -p 8787:8787 -e PASSWORD=mypassword rocker/tidyverse:4.0.0
```

- `-d` Detach (i.e. run as background process).
- `-p 8787:8787` Share a port with the host computer's browser.
- `-e PASSWORD=mypassword` Password for logging on to RStudio Server.
- `rocker/tidyverse:4.0.0` Use the `tidyverse` image built on top of R 4.0.0.

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

- Confirm for yourself that it's actually running with `$ docker ps`. (Mac and Windows users should definitely do this because you'll need the container ID shortly.)

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

Example 2: RStudio+ (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.

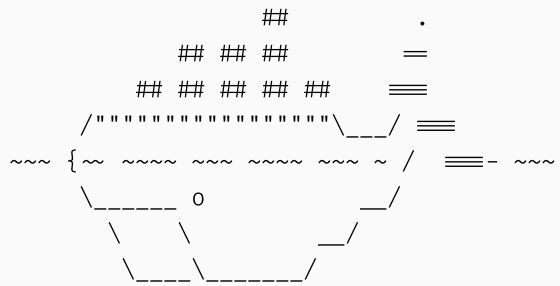
Example 2: RStudio+ (cont.)

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 opened `:8787` port:

- **Linux:** http://localhost:8787
- **Mac/Windows:** Type in `$ docker inspect <containerid> | grep IPAddress` to get your address (see here). Note that this information was also displayed when you first launched your Docker Quickstart Terminal. For example:



```
docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com
```

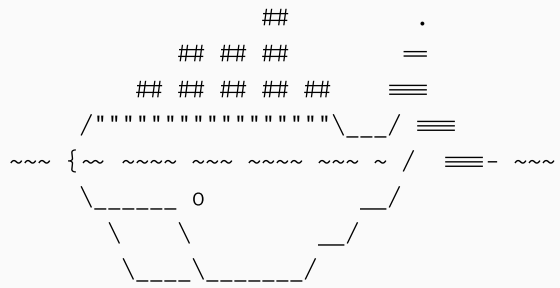
Example 2: RStudio+ (cont.)

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 opened `:8787` port:

- **Linux:** http://localhost:8787
- **Mac/Windows:** Type in `$ docker inspect <containerid> | grep IPAddress` to get your address (see here). Note that this information was also displayed when you first launched your Docker Quickstart Terminal. For example:

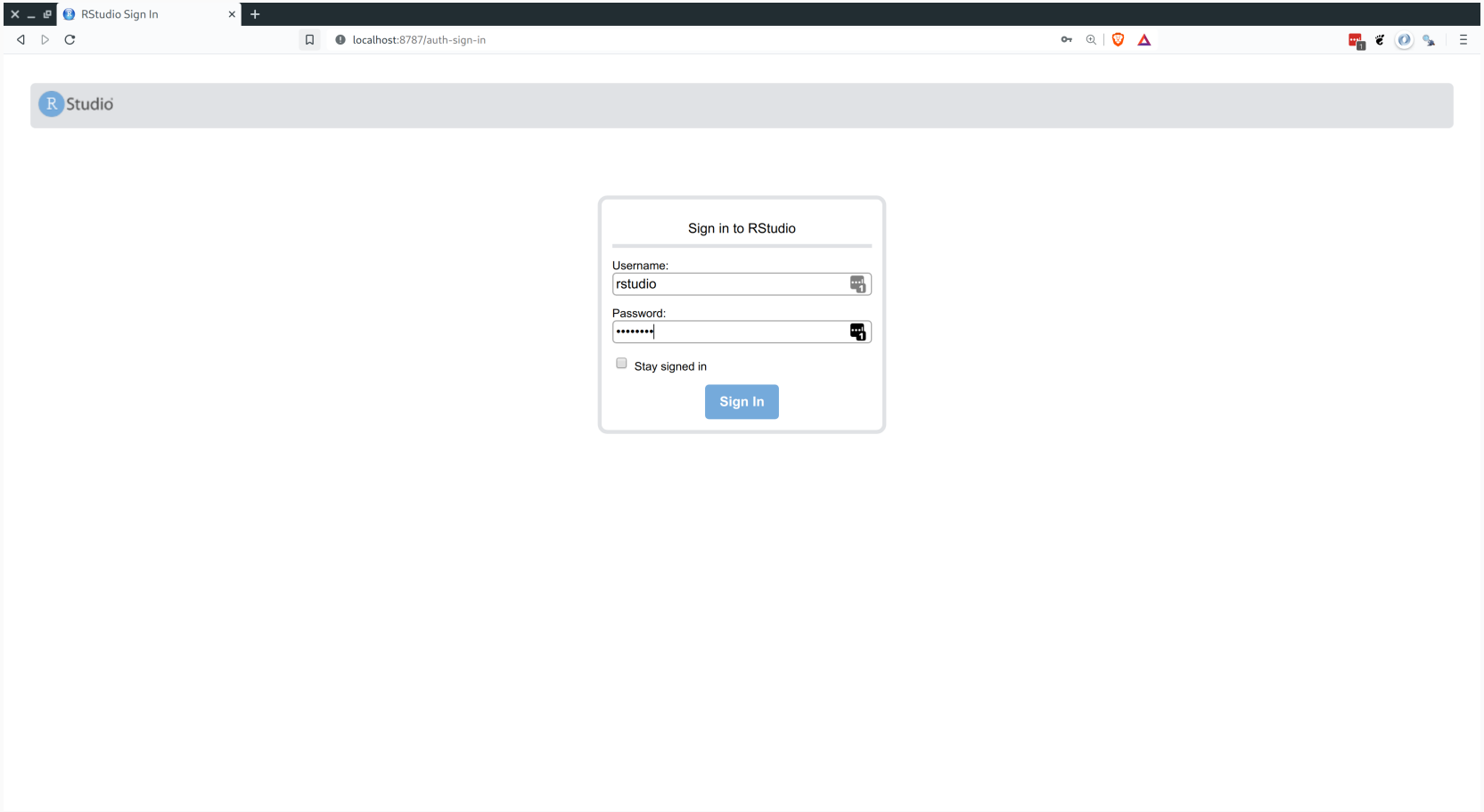


```
docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com
```

So this Mac/Windows user would point their browser to http://192.168.99.100:8787.

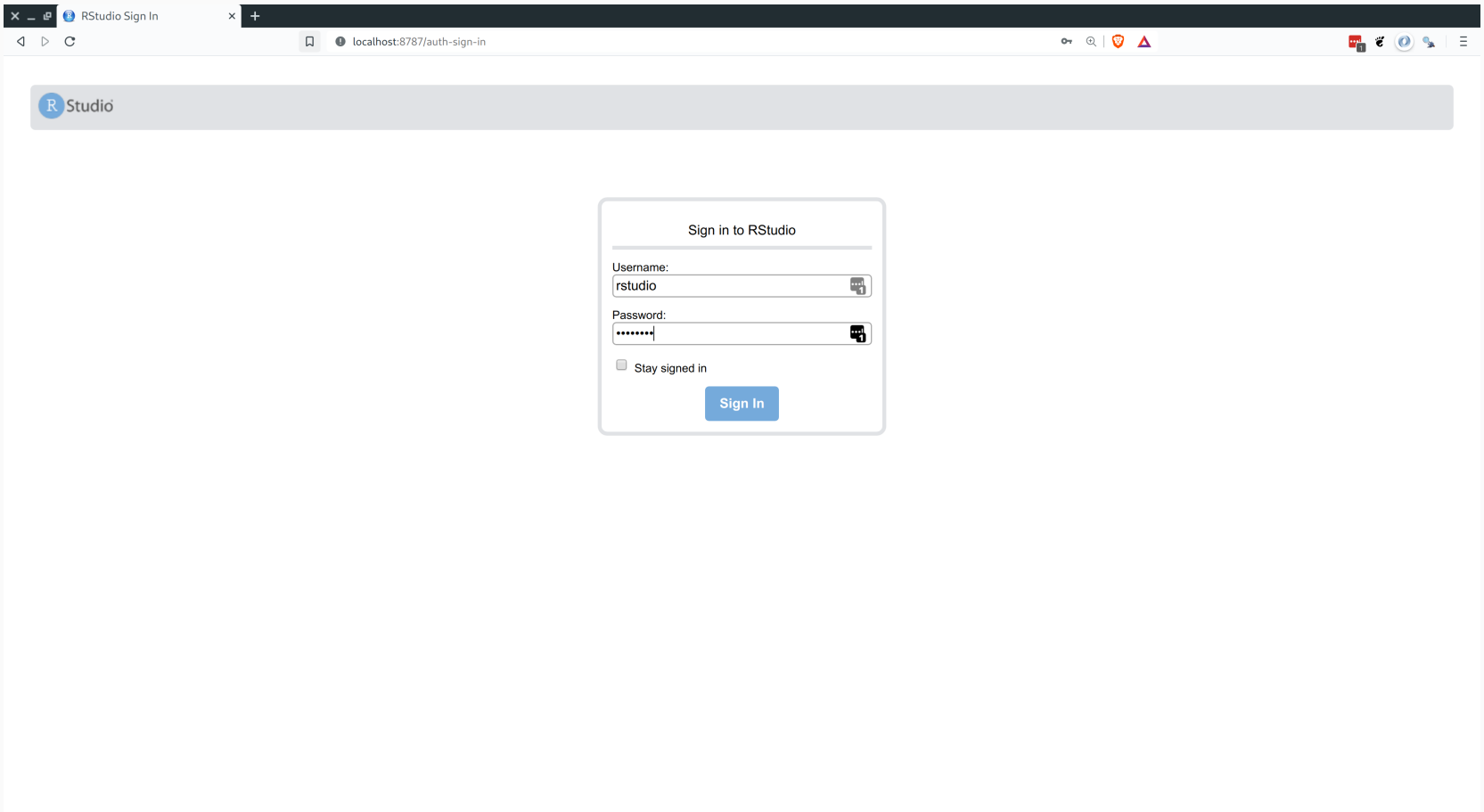
Example 2: RStudio+ (cont.)

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



Example 2: RStudio+ (cont.)

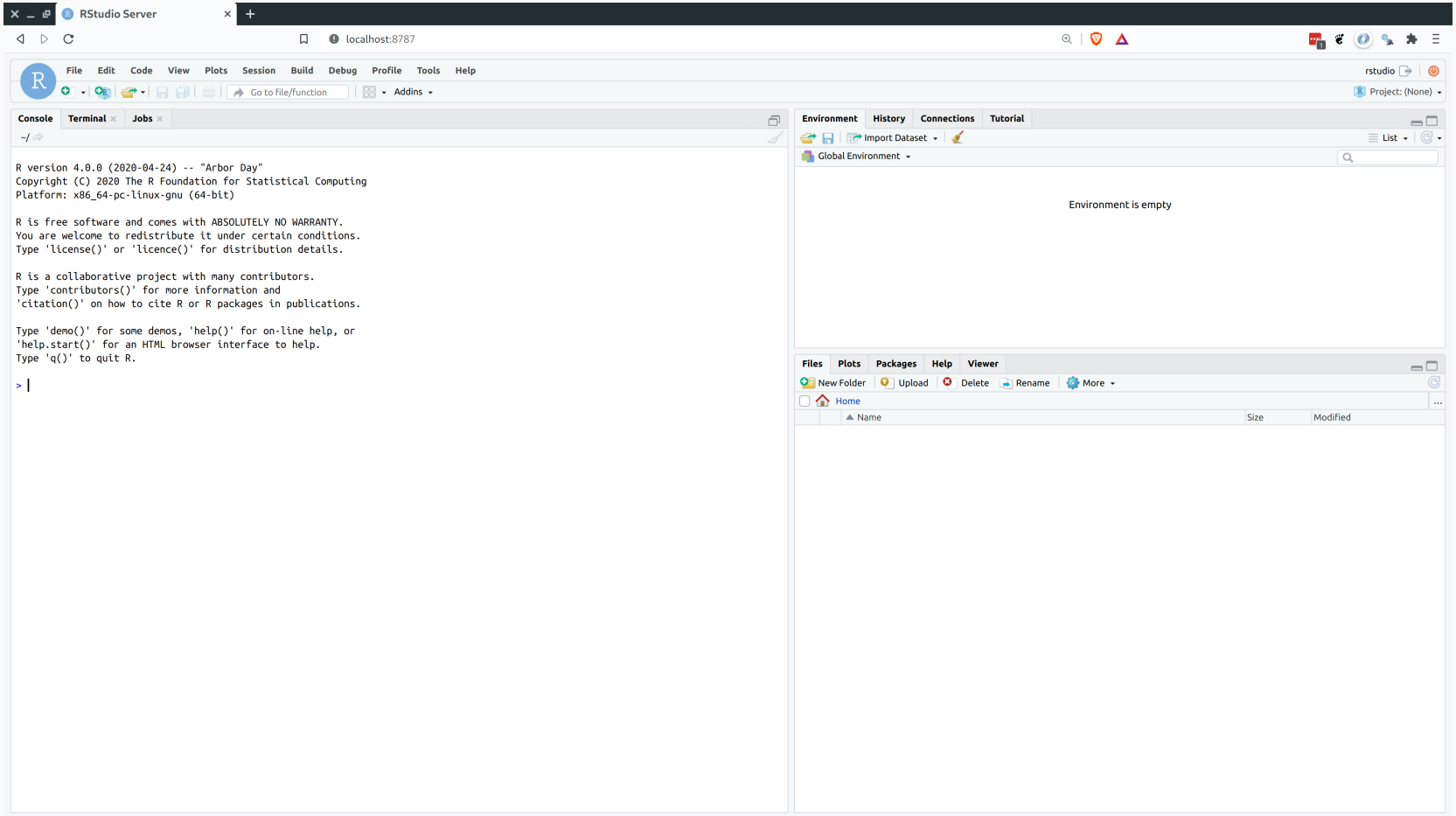
Here's the login-in screen that I see when I navigate my browser to the relevant URL.



Sign in with your "rstudio" + "mypassword" credential combination.

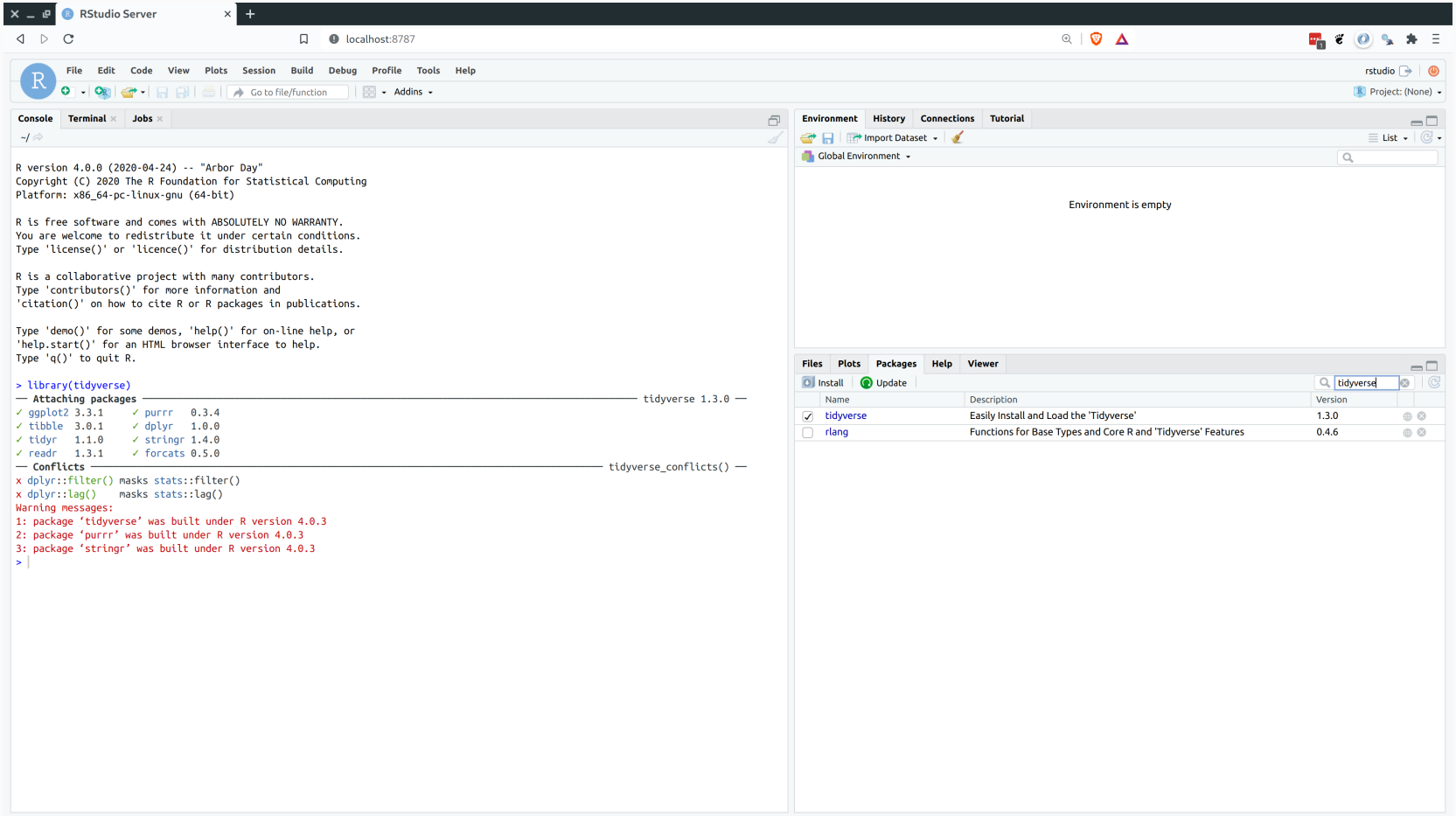
Example 2: RStudio+ (cont.)

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



Example 2: RStudio+ (cont.)

I can also load the **tidyverse** straight away. (We can ignore those warning messages.)



Example 2: RStudio+ (cont.)

To stop this container, open up a new terminal window and grab the container ID with `$ docker ps`. Then run:

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

Example 2: RStudio+ (cont.)

To stop this container, open up a new terminal window and grab the container ID with `$ docker ps`. Then run:

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

Aside: Recall that we instantiated this container as a detached/background process (`-d`).

```
$ 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`.

Example 2: RStudio+ (cont.)

I'll end this example by reiterating the stacked (or *layered*) nature of the Docker workflow.

To prove this, consider what happens what happens when I instantiate the `r-ver:4.0.0` image at the base of Rocker's versioned stack.

Example 2: RStudio+ (cont.)

I'll end this example by reiterating the stacked (or *layered*) nature of the Docker workflow.

To prove this, consider what happens what happens when I instantiate the `r-ver:4.0.0` image at the base of Rocker's versioned stack.

```
~ docker run --rm -it rocker/r-ver:4.0.0
Unable to find image 'rocker/r-ver:4.0.0' locally
4.0.0: Pulling from rocker/r-ver
a4a2a29f9ba4: Already exists
127c9761dcba: Already exists
d13bf203e905: Already exists
4039240d2e0b: Already exists
b5614fff8d2d: Already exists
1f21ceff6ca0: Already exists
Digest: sha256:98dc51733886af5d98cd3f854ceaf142ca25c96830e78011cc0aba651e2eb7e6
Status: Downloaded newer image for rocker/r-ver:4.0.0

R version 4.0.0 (2020-04-24) -- "Arbor Day"
Copyright (C) 2020 The R Foundation for Statistical Computing
```

Example 2: RStudio+ (cont.)

I'll end this example by reiterating the stacked (or *layered*) nature of the Docker workflow.

To prove this, consider what happens what happens when I instantiate the `r-ver:4.0.0` image at the base of Rocker's versioned stack.

TL;DR I am immediately taken into a running R 4.0.0 container.

- All those messages — `a4a2a29f9ba4: Already exists` etc. — are Docker confirming that it already has the necessary layers for building this (parent) container.
- No need to download or build any new layers.

Example 2: RStudio+ (cont.)

I'll end this example by reiterating the stacked (or *layered*) nature of the Docker workflow.

To prove this, consider what happens what happens when I instantiate the `r-ver:4.0.0` image at the base of Rocker's versioned stack.

TL;DR I am immediately taken into a running R 4.0.0 container.

- All those messages — `a4a2a29f9ba4: Already exists` etc. — are Docker confirming that it already has the necessary layers for building this (parent) container.
- No need to download or build any new layers.

This layered approach is not unique to the Rocker stack. It is integral to Docker's core design.

- Cache existing layers. Only (re)build what we have to do.
- Modularity reduces build times, makes containers easy to share and customize.

Example 2: RStudio+ (cont.)

I'll end this example by reiterating the stacked (or *layered*) nature of the Docker workflow.

To prove this, consider what happens what happens when I instantiate the `r-ver:4.0.0` image at the base of Rocker's versioned stack.

TL;DR I am immediately taken into a running R 4.0.0 container.

- All those messages — `a4a2a29f9ba4: Already exists` etc. — are Docker confirming that it already has the necessary layers for building this (parent) container.
- No need to download or build any new layers.

This layered approach is not unique to the Rocker stack. It is integral to Docker's core design.

- Cache existing layers. Only (re)build what we have to do.
- Modularity reduces build times, makes containers easy to share and customize.

We'll return to these ideas at the end of the lecture.

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`.

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
```

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`.

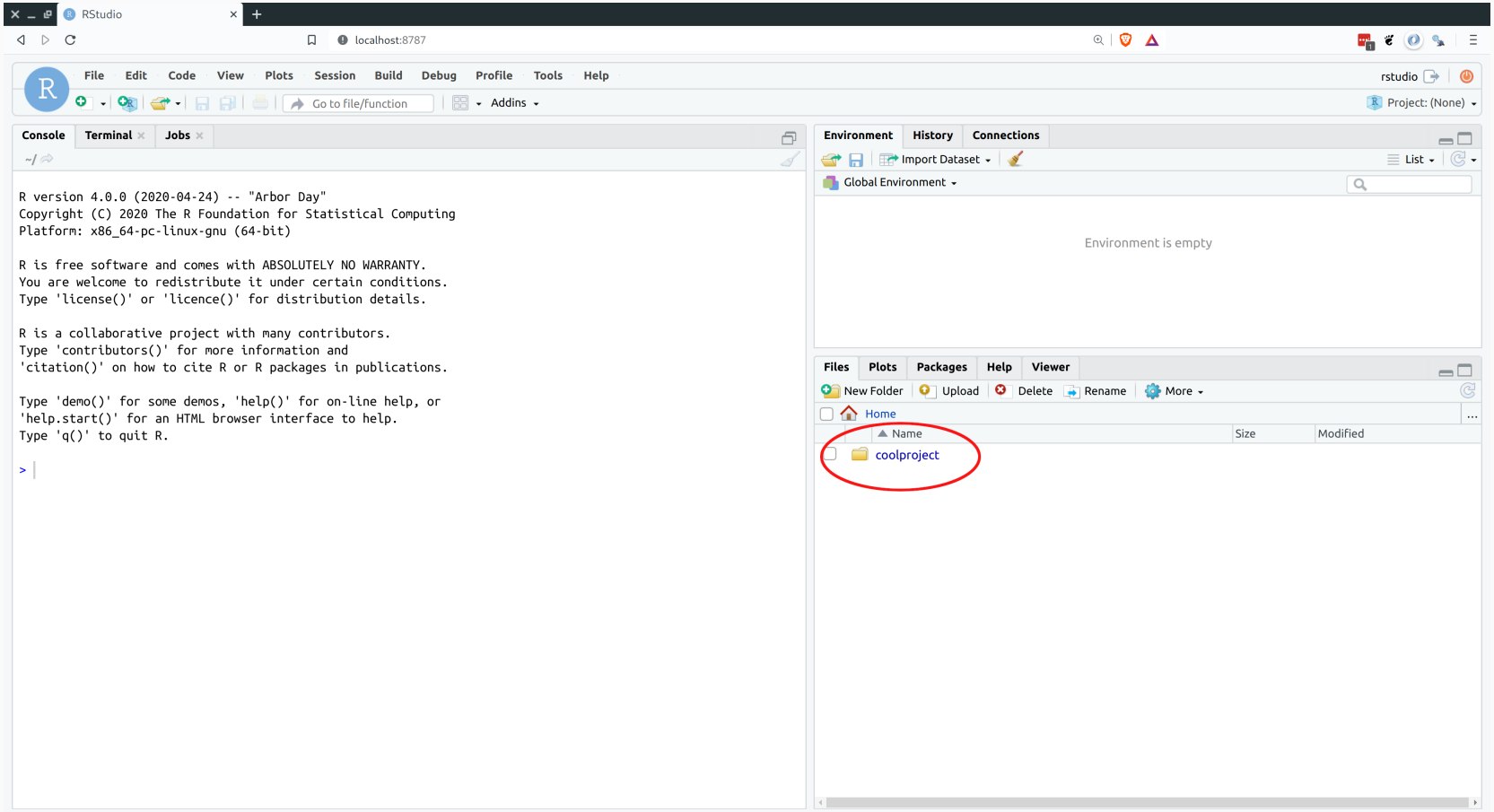
For example, say I have a folder on my computer located at `/home/grant/coolproject`. I can make this available to my "tidyverse" container by running:

```
$ 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?

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).

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).

We have to be be specific about mounting under the user's home directory, because RStudio Server limits how and where users can access files. (This is a security feature that we'll revisit in the next lecture on cloud computing.)

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).

We have to be be specific about mounting under the user's home directory, because RStudio Server limits how and where users can access files. (This is a security feature that we'll revisit in the next lecture on cloud computing.)

OTOH the `/coolproject` directory name is entirely optional. Call it whatever you want... though using the same name as the linked computer directory obviously avoids confusion.

- Similarly, you're free to add a couple of parent directories. I could have used `-v /home/grant/coolproject:/home/rstudio/parentdir1/parentdir2/coolproject` and it would have worked fine.

Choosing the RHS mount point (cont.)

Choosing a specific RHS mount point is less important for non-RStudio+ containers.

Still, be aware that the `/home/rstudio` path won't work for our r-base container from earlier.

- Reason: There's no "rstudio" user. (Fun fact: When you run an r-base container you are actually logged in as root.)

Choosing the RHS mount point (cont.)

Choosing a specific RHS mount point is less important for non-RStudio+ containers.

Still, be aware that the `/home/rstudio` path won't work for our r-base container from earlier.

- Reason: There's no "rstudio" user. (Fun fact: When you run an r-base container you are actually logged in as root.)

For non-Rstudio+ containers, I recommend a general strategy of mounting external volumes on the dedicated `/mnt` directory that is standard on Linux.[1] For example:

```
$ docker run -it --rm -v /home/grant/coolproject:/mnt/coolproject r-base /bin/bash
root@958d28472eb0:/# cd /mnt/coolproject/
root@958d28472eb0:/mnt/coolproject# R
```

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

Next steps

Further reading

- Rocker website
- Docker documentation
- Using Docker for Data Science (Very thorough walkthrough, with a focus on composing your own Dockerfiles from scratch.)
- ROpenSci Docker Tutorial (Another detailed and popular tutorial, albeit outdated in parts.)

Next class: Cloud computing with Google Compute Engine