

# 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. Cleaning up
6. Conclusions

# 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

# 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**.<sup>\*</sup>

<sup>\*</sup> 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**.<sup>\*</sup>

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

<sup>\*</sup> 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

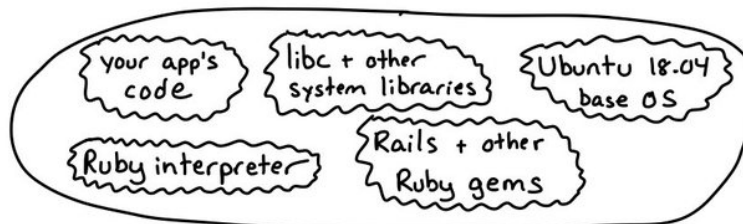
containers package  
EVERY dependency  
together



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:



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!

Credit: [Julia Evans](#). (Buy the [zine](#)!)

# Quick terminology clarification

*Dockerfile* ~ The list of layers and instructions for building a Docker image. (The sheet music.)

*Image* ~ This is the tarball that we talked about on the previous two slides. (MP3 file.)

*Container* ~ A container is a running instance of an image. (Song playing on my computer.)

# Quick terminology clarification

*Dockerfile* ~ The list of layers and instructions for building a Docker image. (The sheet music.)

*Image* ~ This is the tarball that we talked about on the previous two slides. (MP3 file.)

*Container* ~ A container is a running instance of an image. (Song playing on my computer.)

**Analogy:** Think of the Dockerfile as a piece of sheet music, which tells us everything we need to play a song (key, instruments, chords, tempo, etc.) The image is a recording of the music that perfectly reflects the sheet music (e.g. an MP3 file). Finally, a container is a playing instance of that file (maybe on my phone, maybe through my home speakers, etc.)



# 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.<sup>1</sup> For example, you could download and run a **Julia container** right now if you so wished.

<sup>1</sup> It's possible to build a Docker image on top of proprietary software (**example**). But license restrictions make this complicated. I've rarely seen 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.<sup>1</sup> 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).

<sup>1</sup> It's possible to build a Docker image on top of proprietary software ([example](#)). But license restrictions make this complicated. I've rarely seen 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
```

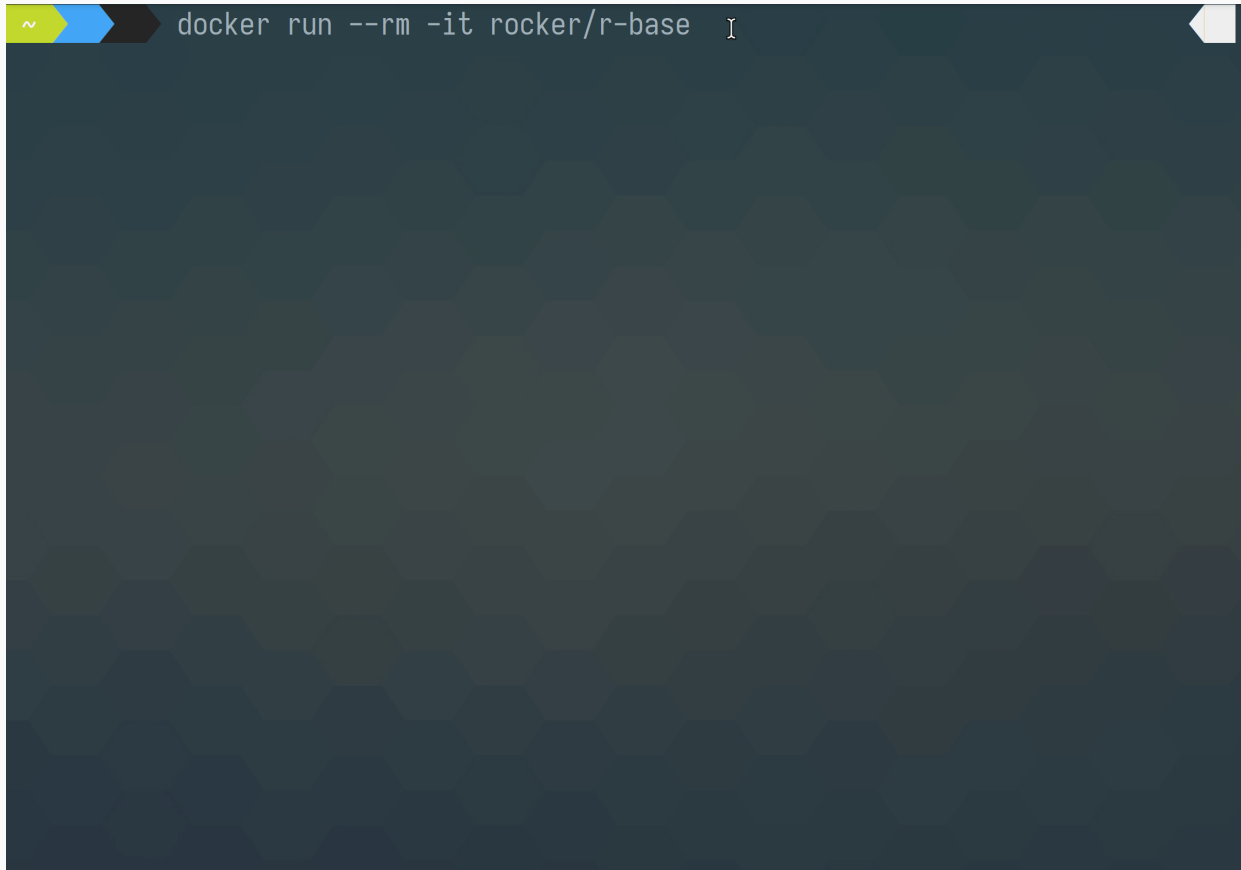
This will take a little while to download the first time (GIF on next slide). But the container will be ready and waiting for immediate deployment on your system thereafter.

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 free 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 free 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 free 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
<a href="#">r-ver</a>	Version-stable base R & src build tools	0B 16 layers	docker pulls 584k
<a href="#">rstudio</a>	Adds rstudio	355.1MB 21 layers	docker pulls 5.7M
<a href="#">tidyverse</a>	Adds tidyverse & devtools	0B 26 layers	docker pulls 2M
<a href="#">verse</a>	Adds tex & publishing-related packages	1.2GB 30 layers	docker pulls 723k
<a href="#">geospatial</a>	Adds geospatial packages on top of 'verse'	0B 32 layers	docker pulls 249k
<a href="#">shiny</a>	Adds shiny server on top of 'r-ver'	0B 13 layers	docker pulls 938k
<a href="#">shiny-verse</a>	Adds tidyverse packages on top of 'shiny'	0B 14 layers	docker pulls 177k
<a href="#">binder</a>	Adds requirements to 'geospatial' to run repositories on <a href="#">mybinder.org</a>	0B 45 layers	docker pulls 74k
<a href="#">ml</a>	Adds python and Tensorflow to 'tidyverse'	0B 47 layers	docker pulls 29k
<a href="#">ml-verse</a>	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
<a href="#">r-ver</a>	Version-stable base R & src build tools	0B 16 layers	docker pulls 584k
<a href="#">rstudio</a>	Adds rstudio	355.1MB 21 layers	docker pulls 5.7M
<a href="#">tidyverse</a>	Adds tidyverse & devtools	0B 26 layers	docker pulls 2M
<a href="#">verse</a>	Adds tex & publishing-related packages	1.2GB 30 layers	docker pulls 723k
<a href="#">geospatial</a>	Adds geospatial packages on top of 'verse'	0B 32 layers	docker pulls 249k
<a href="#">shiny</a>	Adds shiny server on top of 'r-ver'	0B 13 layers	docker pulls 938k
<a href="#">shiny-verse</a>	Adds tidyverse packages on top of 'shiny'	0B 14 layers	docker pulls 177k
<a href="#">binder</a>	Adds requirements to 'geospatial' to run repositories on <a href="https://mybinder.org">mybinder.org</a>	0B 45 layers	docker pulls 74k
<a href="#">ml</a>	Adds python and Tensorflow to 'tidyverse'	0B 47 layers	docker pulls 29k
<a href="#">ml-verse</a>	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=pswd123 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=pswd123` 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=pswd123 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=pswd123` 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=pswd123 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=pswd123` 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 IP 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:

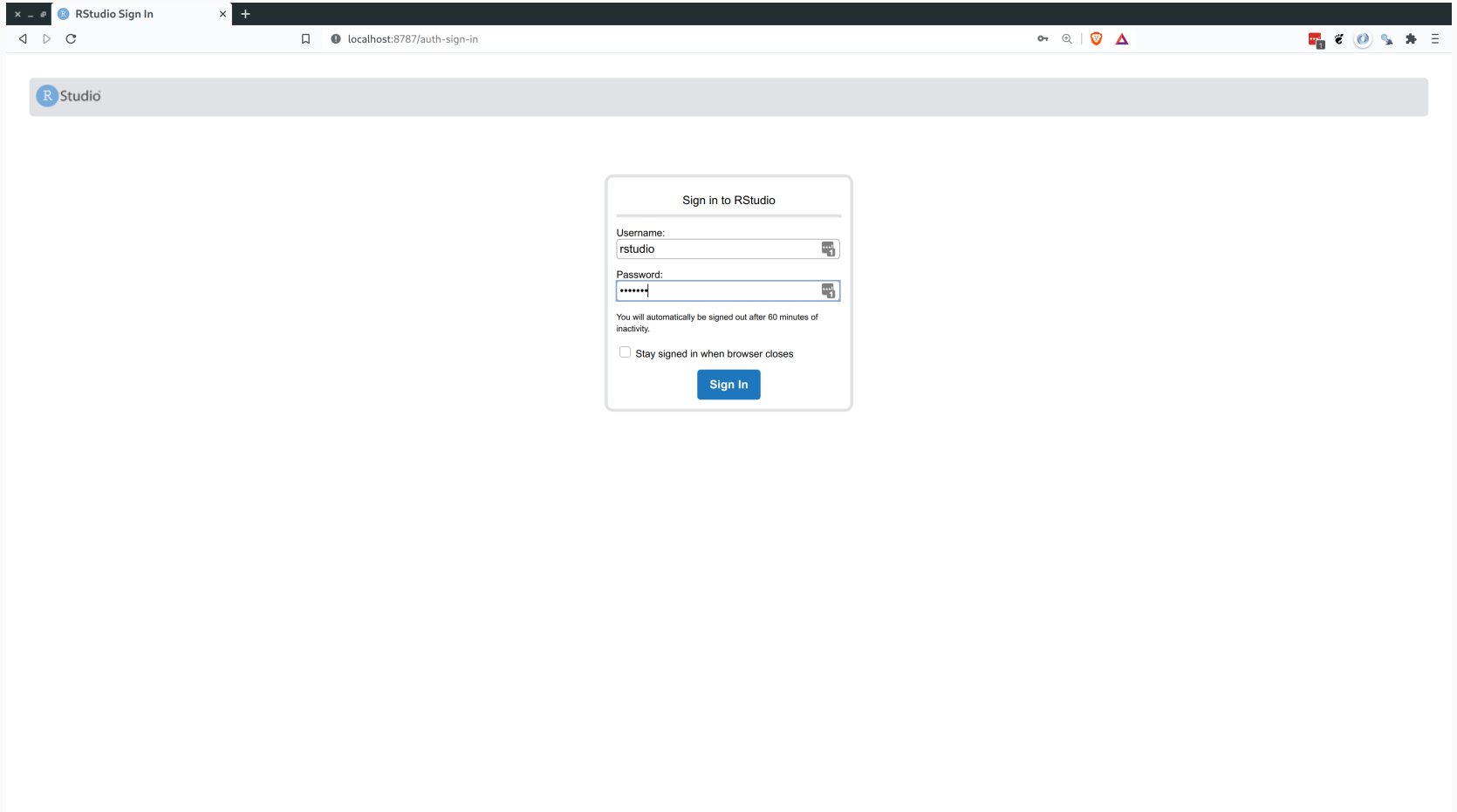
- **Linux:** <http://localhost:8787>
- **Mac/Windows:** Type in `$ docker inspect <containerid> | grep IPAddress` to get your IP address (see [here](#)). Note that this information was also displayed when you first launched your Docker Quickstart Terminal. For example:

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.



The screenshot shows a web browser window with the title "RStudio Sign In". The address bar displays "localhost:8787/auth-sign-in". The browser's toolbar includes navigation buttons (back, forward, refresh), a search icon, and several extension icons. Below the browser window, a grey header bar contains the RStudio logo and the word "Studio". The main content area is white and features a centered login form titled "Sign in to RStudio". The form contains two input fields: "Username:" with the text "rstudio" and "Password:" with masked characters "\*\*\*\*\*". Below these fields, a message states "You will automatically be signed out after 60 minutes of inactivity." and there is a checkbox labeled "Stay signed in when browser closes". At the bottom of the form is a blue "Sign In" button.

Sign in to RStudio

Username:  
rstudio

Password:  
\*\*\*\*\*

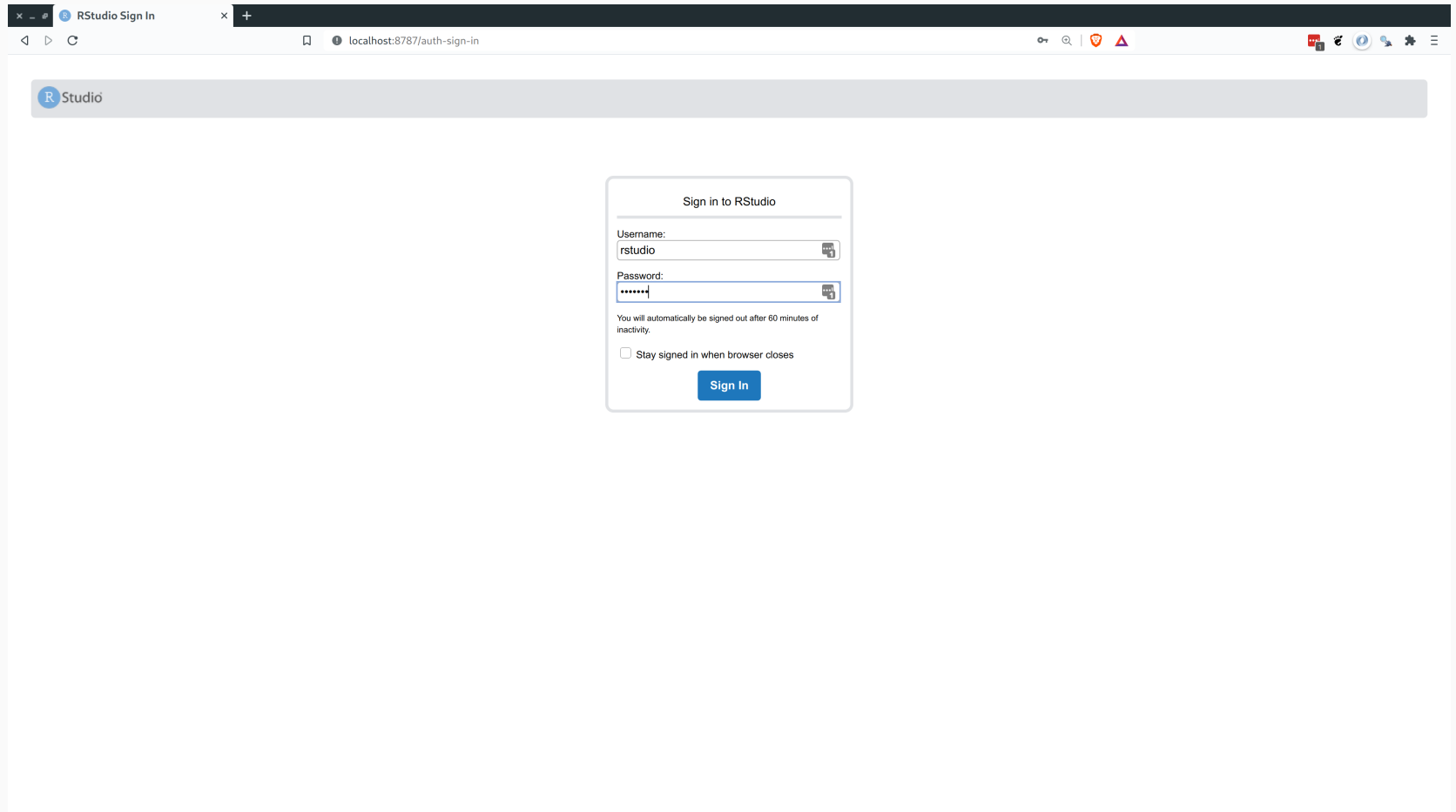
You will automatically be signed out after 60 minutes of inactivity.

☐ Stay signed in when browser closes

Sign In

# Example 2: RStudio+ (cont.)

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

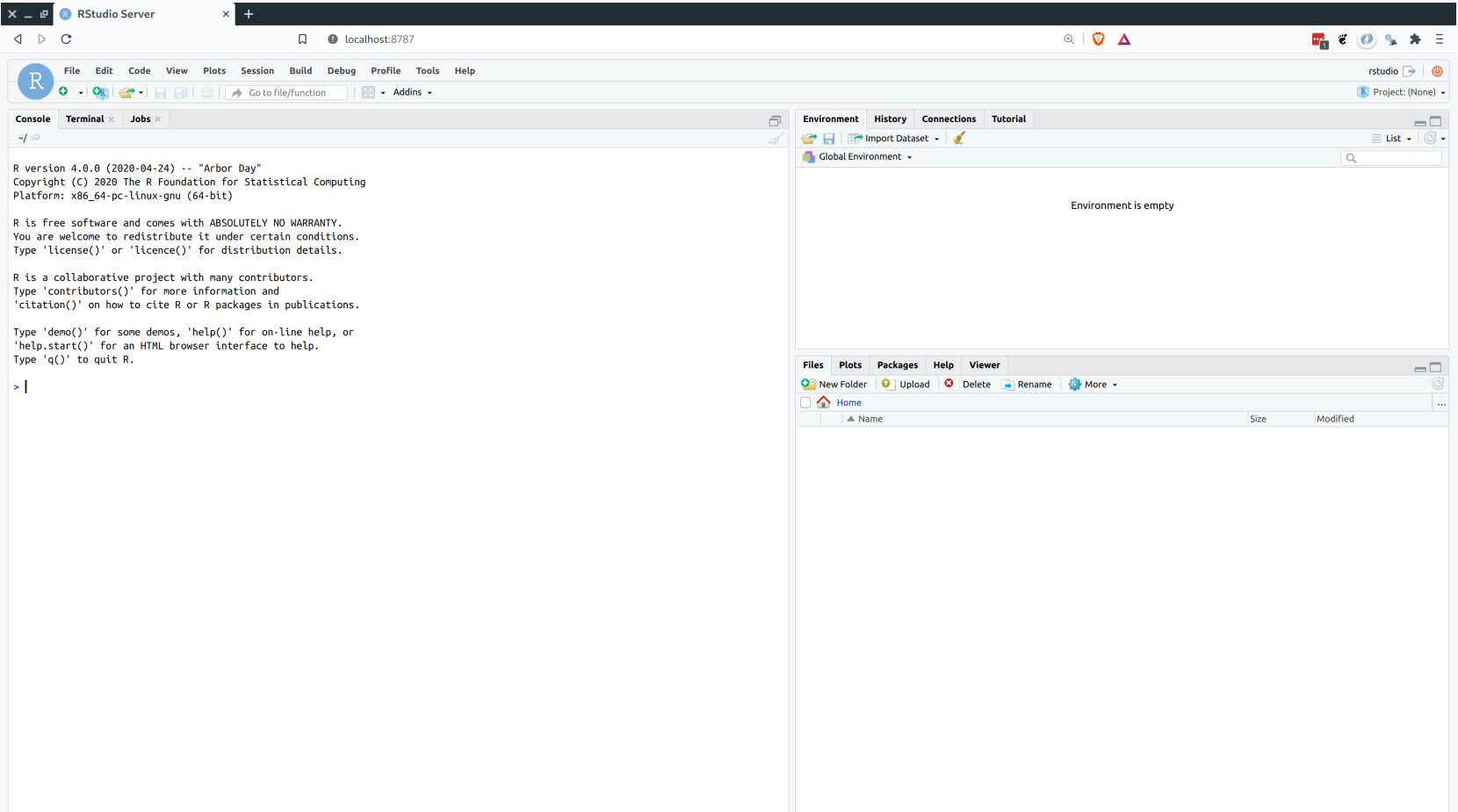


The screenshot shows a web browser window with the title "RStudio Sign In". The address bar displays "localhost:8787/auth-sign-in". The page features a light gray header with the RStudio logo. In the center, there is a white box titled "Sign in to RStudio" containing a login form. The form has two input fields: "Username:" with the value "rstudio" and "Password:" with masked characters "\*\*\*\*\*". Below the password field, a message states "You will automatically be signed out after 60 minutes of inactivity." and there is a checkbox labeled "Stay signed in when browser closes" which is currently unchecked. A blue "Sign In" button is positioned at the bottom of the form.

Sign in with your "rstudio" + "pswd123" 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.)

The screenshot shows the RStudio Server interface. The top bar indicates the connection to 'localhost:8787'. The main window is divided into several panes:

- Console:** Displays the R version (4.0.0), copyright information, and the output of the `library(tidyverse)` command. It shows that several packages (ggplot2, purrr, dplyr, tidyr, stringr, readr, forcats) are being attached. There are also conflict messages for `dplyr::filter()` and `dplyr::lag()`, and warning messages indicating that some packages were built under an older version of R (4.0.3).
- Environment:** Shows the 'Global Environment' and states 'Environment is empty'.
- Packages:** A table listing installed and available packages. The 'tidyverse' package is highlighted, showing its description and version (1.3.0).

```
R version 4.0.0 (2020-04-24) -- "Arbor Day"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> library(tidyverse)
— Attaching packages — tidyverse 1.3.0 —
✓ ggplot2 3.3.1    ✓ purrr  0.3.4
✓ tibble  3.0.1    ✓ dplyr  1.0.0
✓ tidyr   1.1.0    ✓ stringr 1.4.0
✓ readr   1.3.1    ✓ forcats 0.5.0
— Conflicts — tidyverse_conflicts() —
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
Warning messages:
1: package 'tidyverse' was built under R version 4.0.3
2: package 'purrr' was built under R version 4.0.3
3: package 'stringr' was built under R version 4.0.3
>
```

Name	Description	Version
<input checked="" type="checkbox"/> tidyverse	Easily Install and Load the 'Tidyverse'	1.3.0
<input type="checkbox"/> rlang	Functions for Base Types and Core R and 'Tidyverse' Features	0.4.6



# Example 2: RStudio+ (cont.)

To stop this container, open up a new terminal window. Grab the container ID if you've forgotten it with `$ docker ps`. Then run:

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

# Example 2: RStudio+ (cont.)

To stop this container, open up a new terminal window. Grab the container ID if you've forgotten it 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=pswd123 rocker/tidyverse:4.0.0
```

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 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 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 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 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 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 -e PASSWORD=pswd123 rocker/tidyverse:4.0.0
```

# 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`.

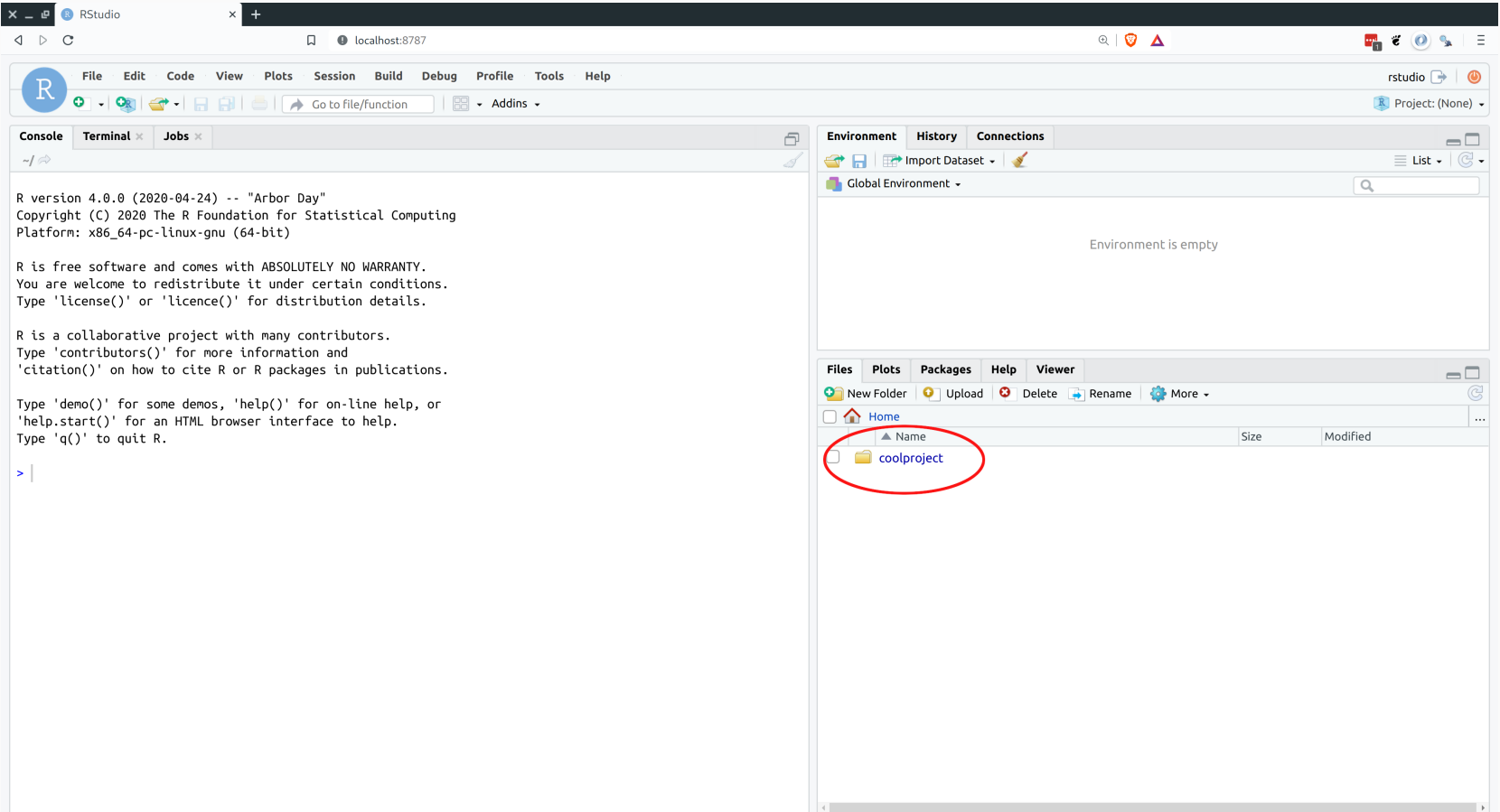
For example, say I have a folder on my computer located at `/home/grant/coolproject`. I can make this available to my "tidyverse" container by running:

```
$ docker run -v /home/grant/coolproject:/home/rstudio/coolproject \  
$ -d -p 8787:8787 -e PASSWORD=pswd123 rocker/tidyverse:4.0.0
```

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.



# pwd

In the previous example, I provided the absolute LHS path to `/home/grant/coolproject`.

The reason is that Docker doesn't understand relative paths for mounting external volumes.

- E.g. I couldn't use `-v ./home/rstudio` or `-v coolproject:home/rstudio`.

# pwd

In the previous example, I provided the absolute LHS path to `/home/grant/coolproject`.

The reason is that Docker doesn't understand relative paths for mounting external volumes.

- E.g. I couldn't use `-v ./home/rstudio` or `-v coolproject:home/rstudio`.

But there *is* a convenient shortcut for mounting the host computer's present working directory: Use ``pwd`` (including the backticks).

```
$ docker run -v `pwd`: /home/rstudio/coolproject \  
$ -d -p 8787:8787 -e PASSWORD=pswd123 rocker/tidyverse:4.0.0
```

# pwd

In the previous example, I provided the absolute LHS path to `/home/grant/coolproject`.

The reason is that Docker doesn't understand relative paths for mounting external volumes.

- E.g. I couldn't use `-v ../home/rstudio` or `-v coolproject:home/rstudio`.

But there *is* a convenient shortcut for mounting the host computer's present working directory: Use ``pwd`` (including the backticks).

```
$ docker run -v `pwd`:/home/rstudio/coolproject \  
$ -d -p 8787:8787 -e PASSWORD=pswd123 rocker/tidyverse:4.0.0
```

This shortcut effectively covers the most common relative path case (i.e. linking a container to our present working directory). You can also specify sub-directories.

- E.g. `-v `pwd`/pics:/home/rstudio`

# 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 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 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. 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
```

# Cleaning up

---

# Docker images

As I keep emphasizing, Docker is fantastic. It allows us to very quickly share and access different software environments, with all the reproducibility and deployment benefits that this entails.

# Docker images

As I keep emphasizing, Docker is fantastic. It allows us to very quickly share and access different software environments, with all the reproducibility and deployment benefits that this entails.

The "downside" of this convenience is that Docker images require disk space.

- For example, the `tidyverse` image that we spun up earlier takes up 2.6 GB.
- Not *huge* given the size of modern hard drives... but you can quickly eat up a good chunk of disk space once you start building Docker images regularly.



# Docker images

As I keep emphasizing, Docker is fantastic. It allows us to very quickly share and access different software environments, with all the reproducibility and deployment benefits that this entails.

The "downside" of this convenience is that Docker images require disk space.

- For example, the `tidyverse` image that we spun up earlier takes up 2.6 GB.
- Not *huge* given the size of modern hard drives... but you can quickly eat up a good chunk of disk space once you start building Docker images regularly.

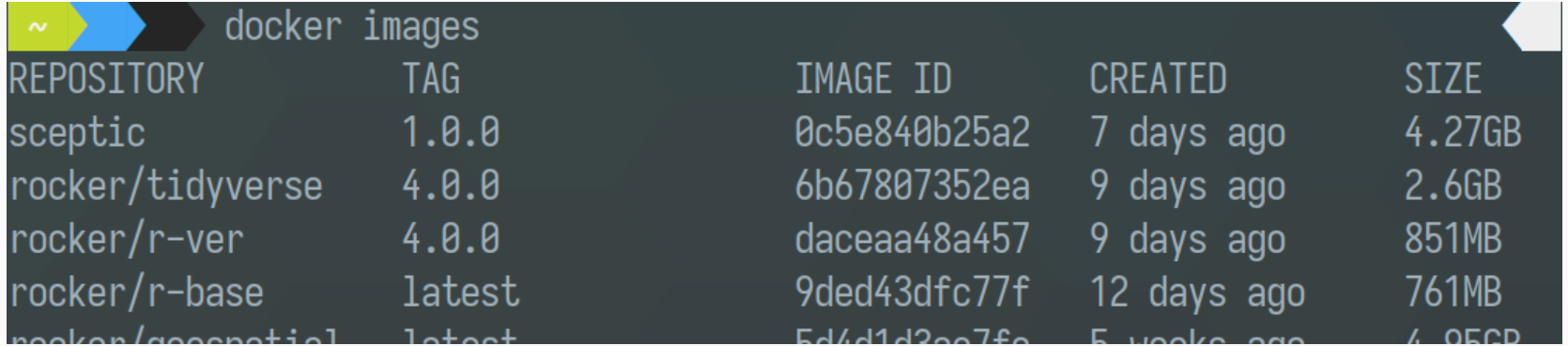
To see a list of the images<sup>1</sup> on your system, simply type:

```
$ docker images
```

<sup>1</sup> **Remember:** Images are distinct from containers.

# Removing images

Running the previous command on my system, here's part of what I see.



```
~ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
sceptic	1.0.0	0c5e840b25a2	7 days ago	4.27GB
rocker/tidyverse	4.0.0	6b67807352ea	9 days ago	2.6GB
rocker/r-ver	4.0.0	daceaa48a457	9 days ago	851MB
rocker/r-base	latest	9ded43dfc77f	12 days ago	761MB
rocker/geoplot	latest	5d4d1d3ee7fe	5 weeks ago	4.95GB

# Removing images

Running the previous command on my system, here's part of what I see.

```
~ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
sceptic	1.0.0	0c5e840b25a2	7 days ago	4.27GB
rocker/tidyverse	4.0.0	6b67807352ea	9 days ago	2.6GB
rocker/r-ver	4.0.0	daceaa48a457	9 days ago	851MB
rocker/r-base	latest	9ded43dfc77f	12 days ago	761MB
rocker/gccnetlib	latest	5d4d1d2ee7fe	5 weeks ago	4.95GB

To remove a particular image (or set of images), we use the `docker rmi <imageid>` command. For example, I could remove both the "rocker/tidyverse" and "rocker/r-ver" images above with:

```
$ docker rmi 6b67807352ea daceaa48a457
```

(Feel free to try this yourself. But don't worry if you'd like to keep the equivalent images on your machine for now.)

# Pruning

Recall that Docker makes heavy use of cached layers to speed up build times.

I mention this because, while the `docker rmi` command normally works great, it doesn't necessarily handle "dangling" images or build caches.

- Basically, intermediate objects that are no longer being used.

This should not matter much for the examples that we've seen today. But, again, these dangling items can waste quite a bit of disk space once you've been building your own Dockerfiles for a while.

# Pruning

Recall that Docker makes heavy use of cached layers to speed up build times.

I mention this because, while the `docker rmi` command normally works great, it doesn't necessarily handle "dangling" images or build caches.

- Basically, intermediate objects that are no longer being used.

This should not matter much for the examples that we've seen today. But, again, these dangling items can waste quite a bit of disk space once you've been building your own Dockerfiles for a while.

To fix this, we use the more aggressive `$ docker <object> prune` command, where `<object>` could be an image, etc. There's also a convenient shorthand for cleaning multiple objects at once:

```
$ docker system prune
```

I frequently use this on my own system. (More on pruning [here](#).)

# Conclusions

---

# Conclusions

Docker makes it easy to configure and share software environments.

- A self-contained "box" with everything needed to run a project or application.
- If it runs on your machine, it will run on my machine.
- Great for testing, reproducibility and deployment.

Terminology analogy

- Dockerfile = sheet music
- Docker image = MP3 recording
- Container = MP3 being played on my computer, phone, etc.

R users are spoilt, thanks to the Rocker Project.

- Easy to build our own Dockerfiles on top of this (or from scratch) if we want.

Example (interactive terminal running base R 4.0.0)

```
$ docker run -it --rm rocker/r-ver:4.0.0
```

# Key commands

- `docker help` list of available commands
- `docker run` downloads (if needed) and runs an image. Useful flags include:
  - `--rm` remove after run
  - `-it` interactive terminal
  - `-v host/path:container/path` share (mount) a directory - `-p 8787:8787` share a browser port: here 8787
- `docker ps` list of currently running containers
- `docker stop <containerids>` stop one or more running containers
- `docker images` list all installed images
- `docker rmi <imageids>` remove one or more images
- `docker system prune` catch all clean-up (stop any running containers, remove any dangling images, etc.)



# Further reading

## Documentation

- [Rocker website](#)
- [Docker documentation](#)

## Tutorials

- [Using R via Rocker](#) (Excellent overview and slidedeck from Dirk Eddelbuettel, one of the originators of the Rocker Project.)
- [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: Google Compute Engine

---