

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
 - R-dev
 - RStudio+
4. Write your own Dockerfiles & images
5. Sharing files with a container
6. Cleaning up
7. Conclusions

Prologue

Install Docker

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

Docker 101

Motivation

Have you ever...

Motivation

Have you ever...

- tried to install a program or run someone else's code, only to be confronted by a bunch of error messages (missing packages, dependencies, etc.)?

Motivation

Have you ever...

- tried to install a program or run someone else's code, only to be confronted by a bunch of error messages (missing packages, dependencies, etc.)?
- shared your code and data with someone else, only for *them* to be confronted by a bunch of error messages (missing packages, dependencies, etc.)?

Motivation

Have you ever...

- tried to install a program or run someone else's code, only to be confronted by a bunch of error messages (missing packages, dependencies, etc.)?
- shared your code and data with someone else, only for *them* to be confronted by a bunch of error messages (missing packages, dependencies, etc.)?
- re-run your *own* code after updating some packages, only to find that it no longer works or the results have changed?

Motivation

Have you ever...

- tried to install a program or run someone else's code, only to be confronted by a bunch of error messages (missing packages, dependencies, etc.)?
- shared your code and data with someone else, only for *them* to be confronted by a bunch of error messages (missing packages, dependencies, etc.)?
- re-run your *own* code after updating some packages, only to find that it no longer works or the results have changed?

Containers are way to solve these and other common software problems.

Motivation

Have you ever...

- tried to install a program or run someone else's code, only to be confronted by a bunch of error messages (missing packages, dependencies, etc.)?
- shared your code and data with someone else, only for *them* to be confronted by a bunch of error messages (missing packages, dependencies, etc.)?
- re-run your *own* code after updating some packages, only to find that it no longer works or the results have changed?

Containers are way to solve these and other common software problems.

Docker

By far the most widely used and best supported container technology. While there are other container platforms around, when I talk about "containers" in this lecture, I'm really talking about **Docker**.

The "container" analogy

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



They provide a standard format for transporting all kinds of goods (TVs, fresh produce, whatever). Moreover, they are stackable and can easily be switched between different transport modes (ship, road, rail).

The "container" analogy

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



They provide a standard format for transporting all kinds of goods (TVs, fresh produce, whatever). Moreover, they are stackable and can easily be switched between different transport modes (ship, road, rail).

Docker containers are the software equivalent.

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

In even simpler terms...

A docker container is just the software equivalent of a box.[†]



- ✓ Standardized shape and form.
- ✓ Everyone can use.
- ✓ "If it runs on your machine, it will run on my machine."

[†] This description (the whole slide, really) is shamelessly stolen from [Dirk Eddelbuettel](#).

In even simpler terms...

A docker container is just the software equivalent of a box.[†]



- ✓ Standardized shape and form.
- ✓ Everyone can use.
- ✓ "If it runs on your machine, it will run on my machine."

More importantly, it allows us to always run code from a pristine, predictable state.

[†] This description (the whole slide, really) is shamelessly stolen from [Dirk Eddelbuettel](#).

Why do we care?

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).

- Examples of academic research projects using Docker for reproducibility [here](#) and [here](#).

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.

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

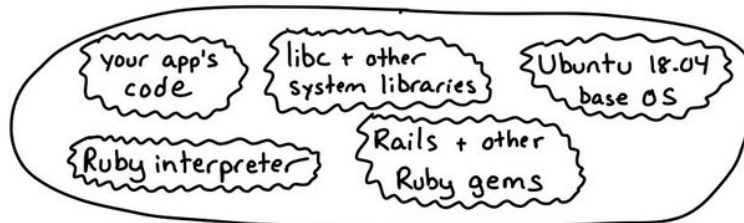
containers package
EVERY dependency
together



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:



how images are built

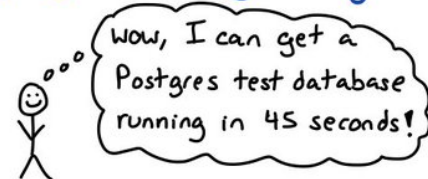
0. start with a base OS
 1. install program + dependencies
 2. configure it how you want
 3. make a tarball of the
WHOLE FILESYSTEM
- (this is what 'docker build' does)

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 sheet music." The list of layers and instructions for building a Docker image.

Image ~ "The MP3 file." This is the tarball that we talked about on the previous two slides.

Container ~ "Song playing on my phone." A container is a running instance of an image.

Quick terminology clarification

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

Image ~ "The MP3 file." This is the tarball that we talked about on the previous two slides.

Container ~ "Song playing on my phone." A container is a running instance of an image.

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). A container is a playing instance of that file (maybe on my phone, maybe through my home speakers, etc.)

Examples

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.[†] For example, you could download and run a **Julia container** right now if you so wished.

[†] 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.[†] 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).

[†] 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.

Example 1: Base R

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
```

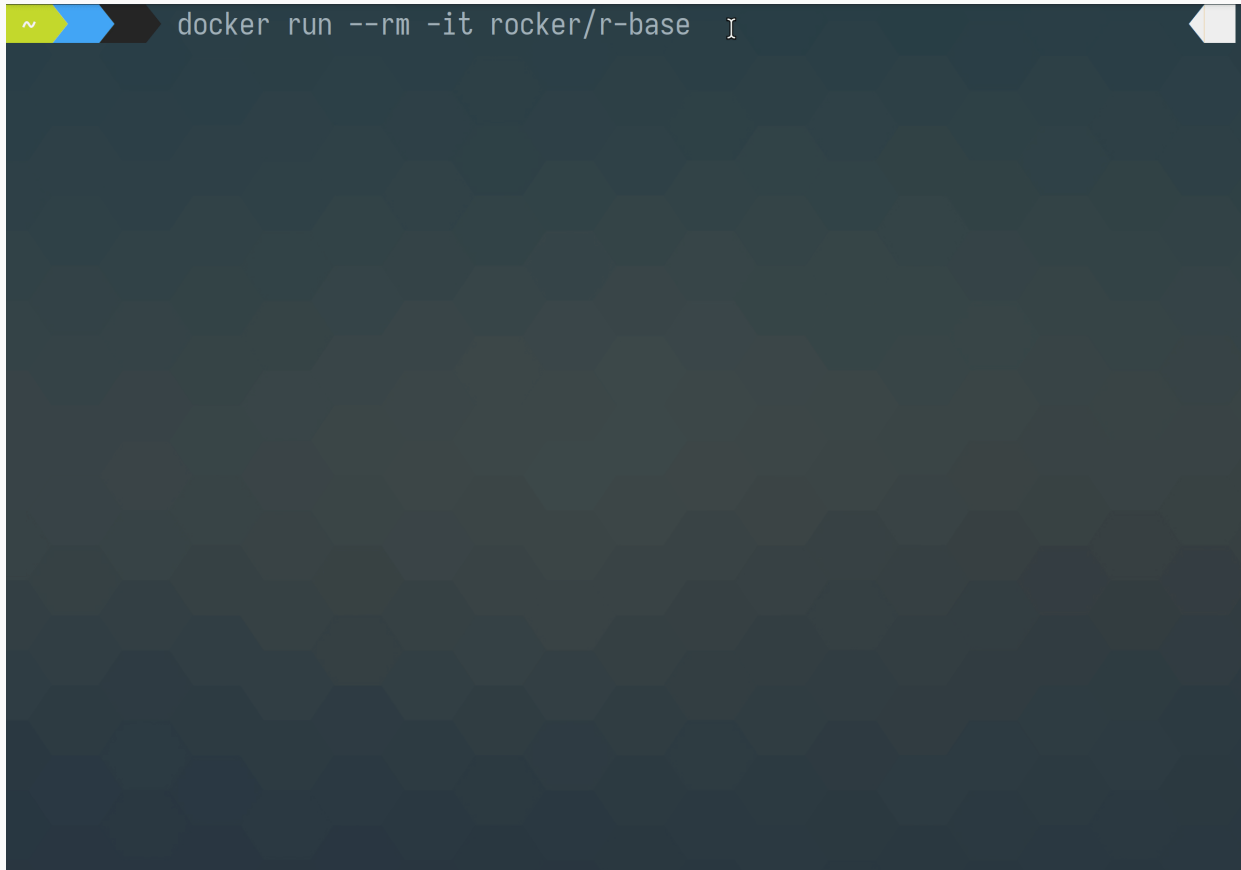
This will take a little while to download the first time (GIF on next slide). But the container will be ready and waiting for immediate deployment on your system thereafter.

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 launches 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: `fcdee074beb`) 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. Do some addition, run a regression on the `mtcars` dataset, etc.

Example 1: Base R (cont.)

Your base R container should have launched directly into R. Feel free to kick the tyres. Do some addition, run a regression on the `mtcars` dataset, 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. Do some addition, run a regression on the `mtcars` dataset, 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: R-dev

Example 2: R-dev

One of Docker's exemplar uses is testing software ahead of time. For example, we can safely test a new version of R in a Docker container before upgrading.

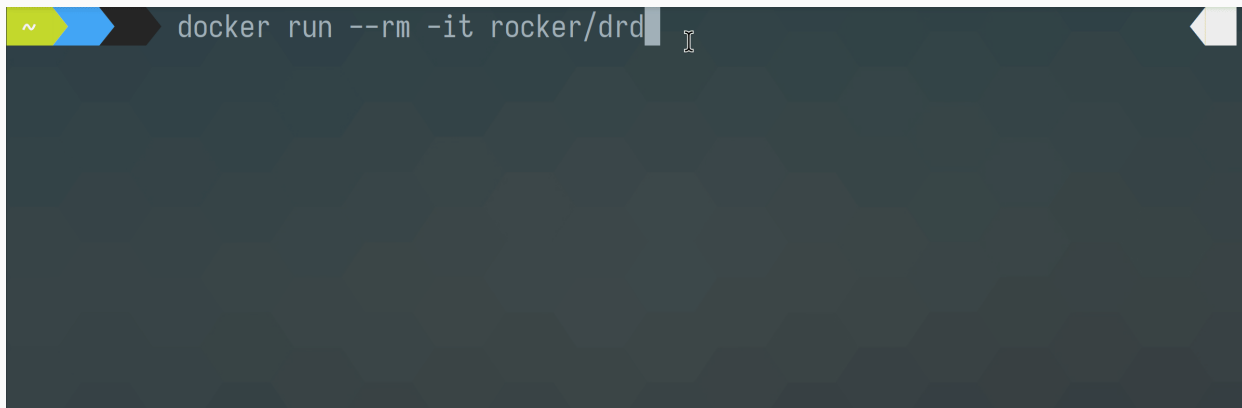
Example 2: R-dev

One of Docker's exemplar uses is testing software ahead of time. For example, we can safely test a new version of R in a Docker container before upgrading.

In that vein, we can also test the development version of R via the `rocker/drd` image.

- At the time of writing this slide, the release of R version 4.1.0 was just a few days away. So, this provides a convenient way to test drive some of the **new features** (a native pipe, lambda functions, etc.)

```
$ docker run --rm -it rocker/drd
```



Example 3: RStudio+

Example 3: 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 3: 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 3: 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 3: 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 3: 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`. (Windows users should definitely do this because you'll need the container ID shortly.)

Example 3: 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`. (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 3: 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.

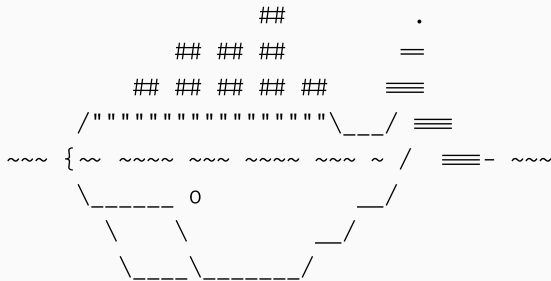
Example 3: RStudio+ (cont.)

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/Mac:** <http://localhost:8787>
- **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 3: 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/Mac:** <http://localhost:8787>
- **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:

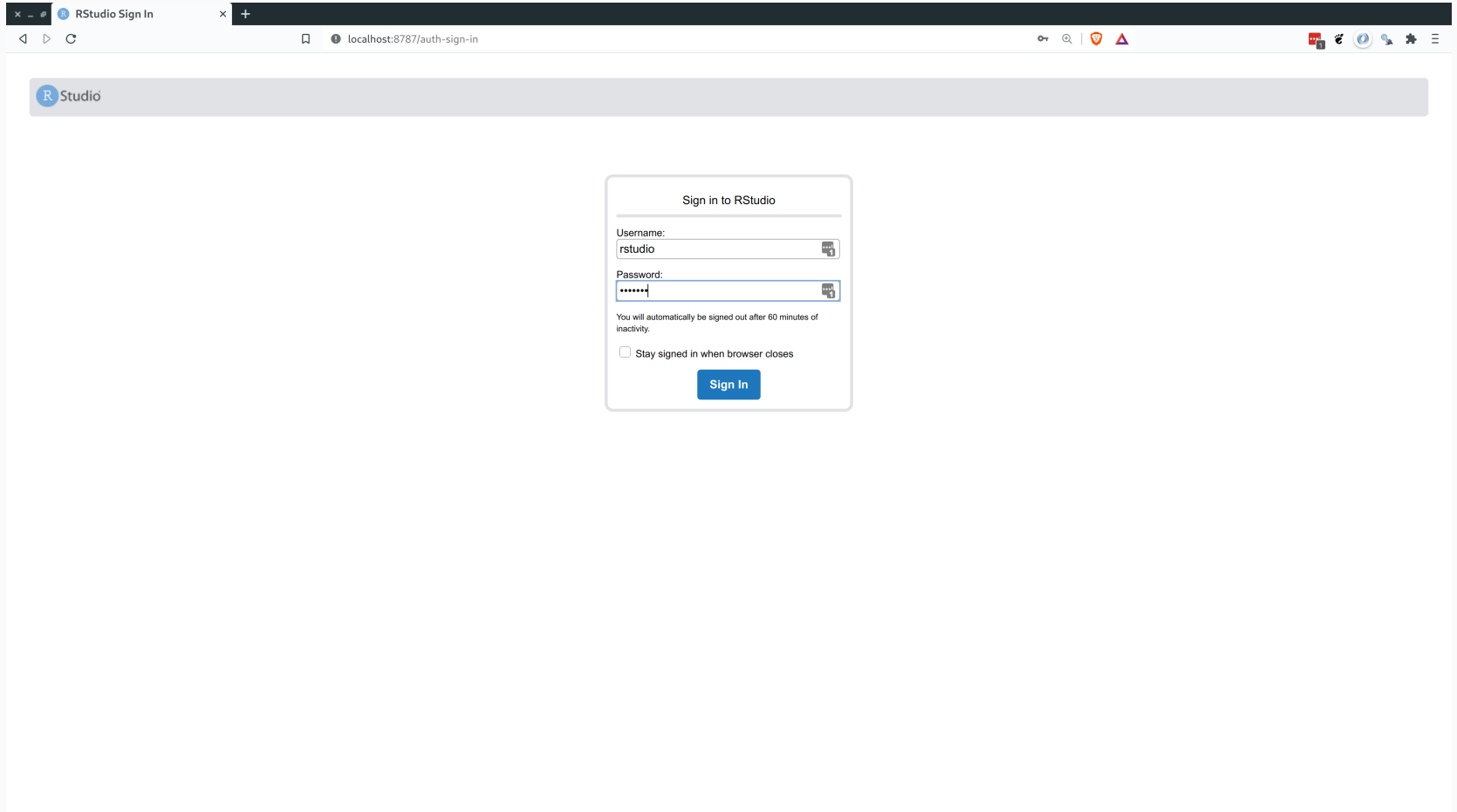
```
## .  
## ## ## =  
## ## ## ## ## ==  
/""""""""""""""""""""\___/ ==  
~~~ { ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ / ==- ~~~  
    \_____o_____/_____  
    \_____\_____/_____  
    \_____\_____/_____
```

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 Windows user would point their browser to <http://192.168.99.100:8787>.

Example 3: RStudio+ (cont.)

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

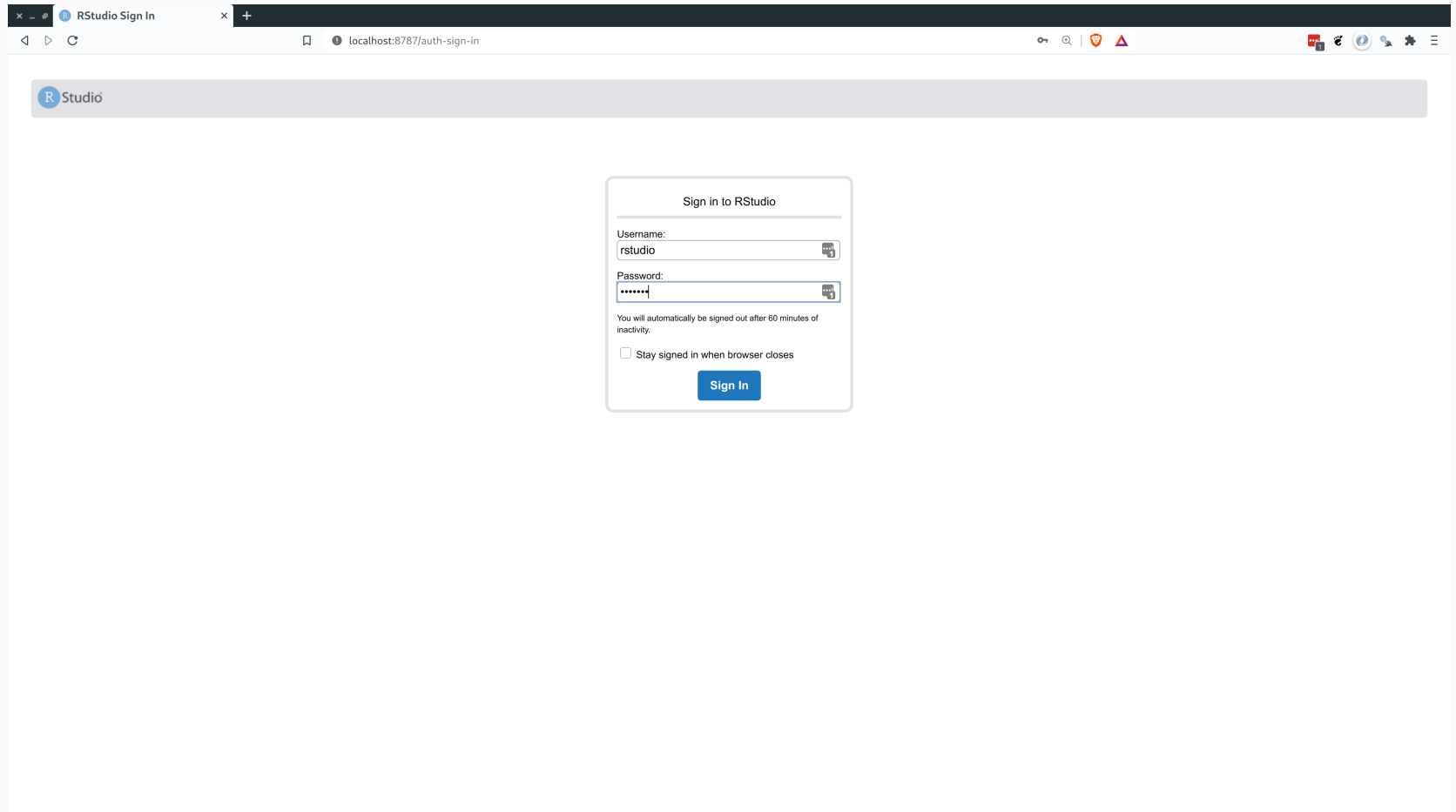


The screenshot shows a web browser window with the title "RStudio Sign In". The address bar displays "localhost:8787/auth-sign-in". The page features a light gray header with the RStudio logo and name. Centered on the page is a white login form with a light gray border. The form is titled "Sign in to RStudio" and contains the following elements:

- A "Username:" label followed by a text input field containing "rstudio".
- A "Password:" label followed by a password input field with masked characters "*****".
- A note: "You will automatically be signed out after 60 minutes of inactivity."
- A checkbox labeled "Stay signed in when browser closes", which is currently unchecked.
- A blue "Sign In" button at the bottom of the form.

Example 3: RStudio+ (cont.)

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

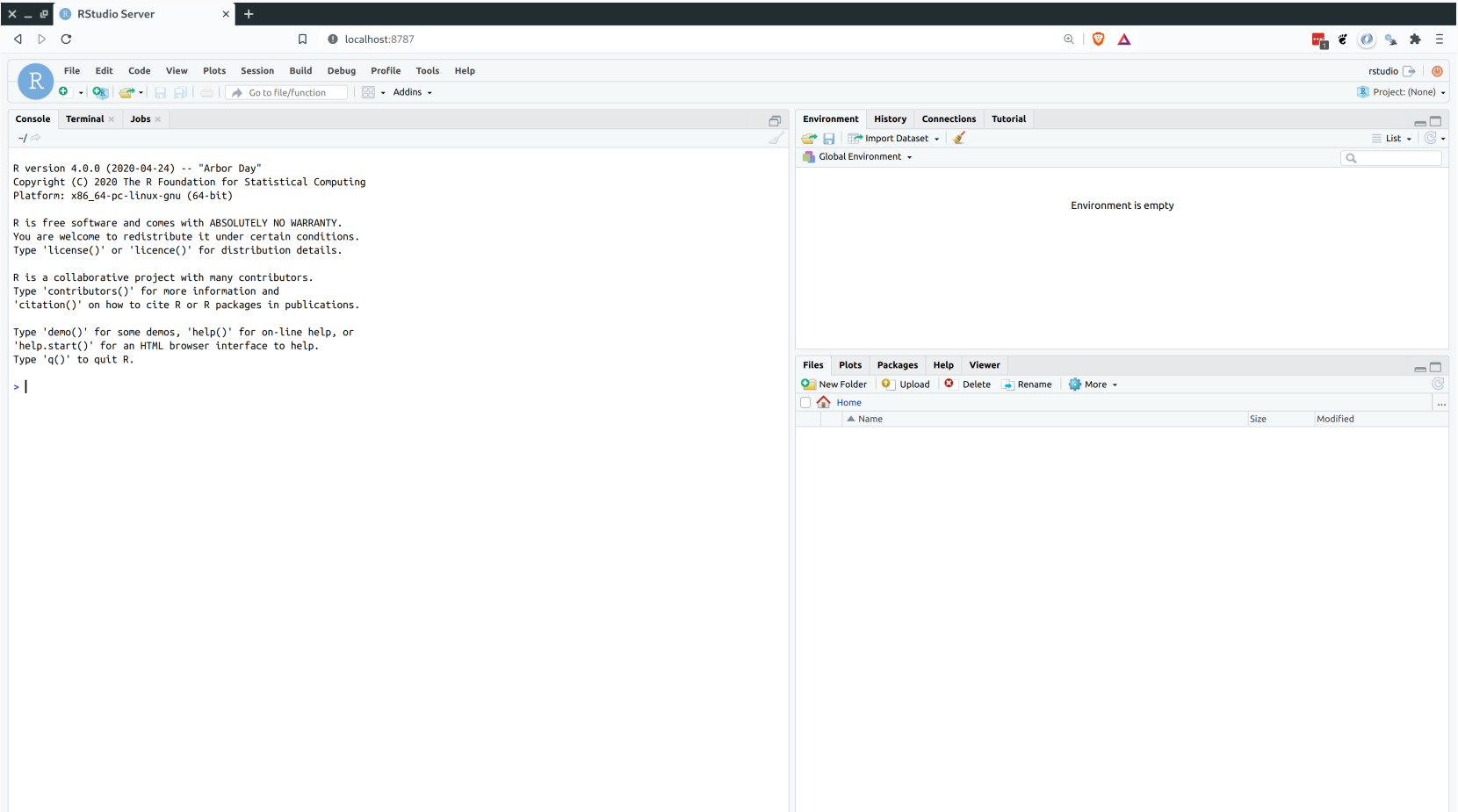


The screenshot shows a web browser window with the title "RStudio Sign In". The address bar displays "localhost:8787/auth-sign-in". The page features a light gray header with the RStudio logo and name. In the center, there is a white box titled "Sign in to RStudio" containing a login form. The form has 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 3: RStudio+ (cont.)

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



Example 3: 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. The left pane contains the R console, which displays the R version (4.0.0), copyright information, and the output of the `library(tidyverse)` command. The right pane shows the 'Environment' tab, which is currently empty. Below the 'Environment' tab is the 'Packages' tab, which lists installed and available packages. The 'tidyverse' package is highlighted, showing its version (1.3.0) and a description. The 'Rlang' package is also listed with version 0.4.6.

```
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 3: RStudio+ (cont.)

To stop this container, you would grab the container ID (i.e. with `$ docker ps`) and then run:

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

Please don't do this yet, however! I want to continue using this running container in the next section.

Example 3: RStudio+ (cont.)

To stop this container, you would grab the container ID (i.e. with `$ docker ps`) and then run:

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

Please don't do this yet, however! I want to continue using this running container in the next section.

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 for yourself later.)

- Everything else would stay the same. You'd still log in at `<IPADDRESS>:8787`, 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 for yourself later.

Example 3: 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 3: 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 3: 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 3: 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 3: 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.

All of which provides a nice segue to our next section...

Write your own Dockerfiles & images

Add to an existing container

The easiest way to start writing our own Docker images is by layering on top of existing containers.

- Remember: Like ogres, Docker is all about the layers.

Add to an existing container

The easiest way to start writing our own Docker images is by layering on top of existing containers.

- Remember: Like *ogres*, Docker is all about the layers.

Let's see a simple example where we add an R library to our `tidyverse:4.0.0` image. First, make sure that the container is still running. You should see something like:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	P
802dbd3841c7	rocker/tidyverse:4.0.0	<code>"/init"</code>	8 minutes ago	Up 8 minutes	0

*(If you don't see something like the above, please re-start your container and then log-in to RStudio Server, using the same steps that we saw *previously*.)*

Add to an existing container

The easiest way to start writing our own Docker images is by layering on top of existing containers.

- Remember: Like ogres, Docker is all about the layers.

Let's see a simple example where we add an R library to our `tidyverse:4.0.0` image. First, make sure that the container is still running. You should see something like:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	P
802dbd3841c7	rocker/tidyverse:4.0.0	<code>"/init"</code>	8 minutes ago	Up 8 minutes	0

(If you don't see something like the above, please re-start your container and then log-in to RStudio Server, using the same steps that we saw [previously](#).)

Once you are in RStudio, install **data.table** like you would any normal R library.

- I'm not going to show this with a GIF, but either use RStudio's library installer or run `install.packages("data.table")`.

Add to an existing container (cont.)

Okay, data.table should now be installed on your running container.

Question: If you stopped your container and restarted it, would data.table still be there?

Add to an existing container (cont.)

Okay, data.table should now be installed on your running container.

Question: If you stopped your container and restarted it, would data.table still be there?

Answer: No!

Add to an existing container (cont.)

Okay, data.table should now be installed on your running container.

Question: If you stopped your container and restarted it, would data.table still be there?

Answer: No! Remember, the whole point of Docker is to always start from the same pristine state.

Add to an existing container (cont.)

Okay, data.table should now be installed on your running container.

Question: If you stopped your container and restarted it, would data.table still be there?

Answer: No! Remember, the whole point of Docker is to always start from the same pristine state.

So, we have to commit these changes as part of a new (different) pristine state. The good news is that this is going to look very similar to our Git workflow.

I'm going to show you how on the next slide to keep everything in one place...

Add to an existing container (cont.)

Step 1: Run `docker ps` to ID the running container. We've already done this, but still...

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	P
802dbd3841c7	rocker/tidyverse:4.0.0	<code>"/init"</code>	8 minutes ago	Up 8 minutes	0

Add to an existing container (cont.)

Step 1: Run `docker ps` to ID the running container. We've already done this, but still...

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	P
802dbd3841c7	rocker/tidyverse:4.0.0	<code>"/init"</code>	8 minutes ago	Up 8 minutes	0

Step 2. Grab the container ID (in my case: `802dbd3841c7`). We're going to use this to commit our changes and create a new Docker image, which I'll call `tidyverse_dt`.

```
$ docker commit -m "tidverse + data.table" 802dbd3841c7 tidyverse_dt:4.0.0
```

Again, this should feel *very* familiar to our Git workflow. We even wrote ourselves a helpful commit message. Note that I added a version tag (i.e. `:4.0.0`). This is optional, but good practice. Here, I'm reminding myself that I've built on top of the R 4.0.0 versioned stack.

Add to an existing container (cont.)

Step 1: Run `docker ps` to ID the running container. We've already done this, but still...

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	P
802dbd3841c7	rocker/tidyverse:4.0.0	<code>"/init"</code>	8 minutes ago	Up 8 minutes	0

Step 2. Grab the container ID (in my case: `802dbd3841c7`). We're going to use this to commit our changes and create a new Docker image, which I'll call `tidyverse_dt`.

```
$ docker commit -m "tidverse + data.table" 802dbd3841c7 tidyverse_dt:4.0.0
```

Again, this should feel *very* familiar to our Git workflow. We even wrote ourselves a helpful commit message. Note that I added a version tag (i.e. `:4.0.0`). This is optional, but good practice. Here, I'm reminding myself that I've built on top of the R 4.0.0 versioned stack.

Step 3. Profit. (There's no step 3. Just confirm for yourself that your image has been created.)

```
$ docker images
```

Adding things outside of R?

For the simple example on the previous slide, we installed a single R package. But the process would work exactly the same if we did any other operations from within R(studio).

- Install multiple packages
- Save datasets, scripts, figures, etc.

Adding things outside of R?

For the simple example on the previous slide, we installed a single R package. But the process would work exactly the same if we did any other operations from within R(studio).

- Install multiple packages
- Save datasets, scripts, figures, etc.

Question: What happens if we want to install/update something outside of R(Studio)? Say, Ubuntu system libraries or another program like Python?

Adding things outside of R?

For the simple example on the previous slide, we installed a single R package. But the process would work exactly the same if we did any other operations from within R(studio).

- Install multiple packages
- Save datasets, scripts, figures, etc.

Question: What happens if we want to install/update something outside of R(Studio)? Say, Ubuntu system libraries or another program like Python?

Answer: It still works exactly the same. You just have to add things through your container's bash shell. Either:

- Launch directly into the shell to start with (remember: [here](#)). Or,
- Use `docker exec` to open the shell of a running container. For example:

```
$ docker exec -it 802dbd3841c7 bash
root@802dbd3841c7:/# apt update -y && apt install htop
root@802dbd3841c7:/# htop ## Show all available CPU cores. Press 'q' to quit.
```

Adding things outside of R?

For the simple example on the previous slide, we installed a single R package. But the process would work exactly the same if we did any other operations from within R(studio).

- Install multiple packages
- Save datasets, scripts, figures, etc.

Question: What happens if we want to install/update something outside of R(Studio)? Say, Ubuntu system libraries or another program like Python?

Answer: It still works exactly the same. You just have to add things through your container's bash shell. Either:

- Launch directly into the shell to start with (remember: [here](#)). Or,
- Use `docker exec` to open the shell of a running container. For example:

```
$ docker exec -it 802dbd3841c7 bash
root@802dbd3841c7:/# apt update -y && apt install htop
root@802dbd3841c7:/# htop ## Show all available CPU cores. Press 'q' to quit.
```

(Obviously, you'd now have to commit this change to add `htop` to your image.)

Aside: Stop your container(s)

Okay, now is a good time to stop your container if you haven't done so already. Grab your container ID and run:

```
$ docker stop <container-id>
```

Alternatively, you can stop all running containers with the following command:

```
$ docker stop $(docker ps -q)
```

Write your own Dockerfile

The interactive approach to building Docker images — i.e. committing changes to a running container — is a convenient way to get up and running quickly. However, at some point you'll probably want to start writing your own `Dockerfiles`.

- Remember: `Dockerfiles` are the "sheet music" of the whole operation. These are simple text files that provide the full set of (shell) instructions for building our Docker images.

Write your own Dockerfile

The interactive approach to building Docker images — i.e. committing changes to a running container — is a convenient way to get up and running quickly. However, at some point you'll probably want to start writing your own `Dockerfiles`.

- Remember: `Dockerfiles` are the "sheet music" of the whole operation. These are simple text files that provide the full set of (shell) instructions for building our Docker images.

There is a whole host of **commands and considerations** for writing `Dockerfiles` — all of which I am going to skip for this lecture. (We simply don't have the time.)

Write your own Dockerfile

The interactive approach to building Docker images — i.e. committing changes to a running container — is a convenient way to get up and running quickly. However, at some point you'll probably want to start writing your own `Dockerfiles`.

- Remember: `Dockerfiles` are the "sheet music" of the whole operation. These are simple text files that provide the full set of (shell) instructions for building our Docker images.

There is a whole host of `commands and considerations` for writing `Dockerfiles` — all of which I am going to skip for this lecture. (We simply don't have the time.)

BUT... I will briefly say that the Docker Project again has our backs with a bunch of `ready-made scripts` for building on and extending their Docker images.

Write your own Dockerfile

The interactive approach to building Docker images — i.e. committing changes to a running container — is a convenient way to get up and running quickly. However, at some point you'll probably want to start writing your own `Dockerfiles`.

- Remember: `Dockerfiles` are the "sheet music" of the whole operation. These are simple text files that provide the full set of (shell) instructions for building our Docker images.

There is a whole host of `commands and considerations` for writing `Dockerfiles` — all of which I am going to skip for this lecture. (We simply don't have the time.)

BUT... I will briefly say that the Rocker Project again has our backs with a bunch of `ready-made scripts` for building on and extending their Docker images.

For example, if we wanted to modify the `r-ver4.0.0` image so that it also included Python, our Dockerfile would be as simple as the following two lines:

```
FROM rocker/r-ver:4.0.0
RUN /rocker_scripts/install_python.sh
```

(Continues on next slide.)

Write your own Dockerfile (cont.)

```
FROM rocker/r-ver:4.0.0  
RUN /rocker_scripts/install_python.sh
```

Write your own Dockerfile (cont.)

```
FROM rocker/r-ver:4.0.0  
RUN /rocker_scripts/install_python.sh
```

Try this yourself by creating a file called `Dockerfile`[†] comprising the above lines.

[†]Every `Dockerfile` is called exactly that. Only one `Dockerfile` is allowed per (sub) directory.

Write your own Dockerfile (cont.)

```
FROM rocker/r-ver:4.0.0  
RUN /rocker_scripts/install_python.sh
```

Try this yourself by creating a file called `Dockerfile`[†] comprising the above lines.

Next, build your Docker image from this `Dockerfile` using the following shell command. I'm going to call my image `r_py` and give it the "4.0.0" version stamp (both choices being optional). **Important:** Make sure that your shell/terminal is in the same directory as the `Dockerfile` when you run this command.

```
$ # docker build --tag <name>:<version> <directory>  
$ docker build --tag r_py:4.0.0 .
```

[†]Every `Dockerfile` is called exactly that. Only one `Dockerfile` is allowed per (sub) directory.

Write your own Dockerfile (cont.)

```
FROM rocker/r-ver:4.0.0
RUN /rocker_scripts/install_python.sh
```

Try this yourself by creating a file called `Dockerfile`[†] comprising the above lines.

Next, build your Docker image from this `Dockerfile` using the following shell command. I'm going to call my image `r_py` and give it the "4.0.0" version stamp (both choices being optional). **Important:** Make sure that your shell/terminal is in the same directory as the `Dockerfile` when you run this command.

```
$ # docker build --tag <name>:<version> <directory>
$ docker build --tag r_py:4.0.0 .
```

This will take a minute to pull everything in. But your `r_py` image will be ready for immediate deployment thereafter, and now includes Python and `reticulate`.

```
$ docker run -it --rm r_py:4.0.0
```

[†]Every `Dockerfile` is called exactly that. Only one `Dockerfile` is allowed per (sub) directory.

Write your own Dockerfile (cont.)

Okay, one last tip about writing your own Dockerfiles. Let's say we wanted to add an R package (e.g. `data.table`) to our `r_py` image at build time. How could we do this?

Write your own Dockerfile (cont.)

Okay, one last tip about writing your own Dockerfiles. Let's say we wanted to add an R package (e.g. `data.table`) to our `r_py` image at build time. How could we do this?

Well, remember that Dockerfiles are (basically) just a set of shell instructions. So we can tell our Dockerfile to install an R package via an appropriate bash command like `Rscript`.[†]

- We also need to pre-pend any bash command with the special Docker verb `RUN`.

[†] We covered `Rscript` back in the [shell lecture](#).

Write your own Dockerfile (cont.)

Okay, one last tip about writing your own Dockerfiles. Let's say we wanted to add an R package (e.g. `data.table`) to our `r_py` image at build time. How could we do this?

Well, remember that Dockerfiles are (basically) just a set of shell instructions. So we can tell our Dockerfile to install an R package via an appropriate bash command like `Rscript`.[†]

- We also need to pre-pend any bash command with the special Docker verb `RUN`.

Our Dockerfile thus becomes:

```
FROM rocker/r-ver:4.0.0
RUN /rocker_scripts/install_python.sh
RUN R -e "install.packages('data.table')"
```

[†] We covered `Rscript` back in the [shell lecture](#).

Write your own Dockerfile (cont.)

Okay, one last tip about writing your own Dockerfiles. Let's say we wanted to add an R package (e.g. `data.table`) to our `r_py` image at build time. How could we do this?

Well, remember that Dockerfiles are (basically) just a set of shell instructions. So we can tell our Dockerfile to install an R package via an appropriate bash command like `Rscript`.[†]

- We also need to pre-pend any bash command with the special Docker verb `RUN`.

Our Dockerfile thus becomes:

```
FROM rocker/r-ver:4.0.0
RUN /rocker_scripts/install_python.sh
RUN R -e "install.packages('data.table')"
```

If you build this image you'll see that it completes almost instantly... because the first two lines (i.e. layers) have already been cached. Clever!

[†] We covered `Rscript` back in the [shell lecture](#).

Automated Dockerfiles

Say that you have an existing research project or repo. Is there an easy way to write a Dockerfile / Docker image based on the contents?

Answer: Yes!

I don't have time to go into details, but automated Docker tools include:

- `containerit`
- `repo2docker`

Check them out. (I have a small `containerit` demo [here](#).)

Docker Hub: Share your Docker images

So, you've written a cool Dockerfile that you want to share with world. What next?

Docker Hub: Share your Docker images

So, you've written a cool Dockerfile that you want to share with world. What next?

You can share your Dockerfiles and images in various ways.

- For my research projects, I add a Dockerfile to the companion GitHub repo. This provides a convenient way for others to reproduce the (potentially complex) computing environment that I used for conducting my analysis. (Example [here](#).)

The most popular way to share Docker images is by hosting them on **Docker Hub**.

- I'm not going to show you how to do that here. But the good news is that it's very straightforward. See [here](#) for a quick walkthrough.

Sharing files with a container

Prep: Stop all running containers

This next section is all about sharing files and folders between your computer and a container. To avoid unexpected behaviour, it would be best to stop all running containers before proceeding.

```
$ docker stop $(docker ps -q)
```

Prep: Stop all running containers

This next section is all about sharing files and folders between your computer and a container. To avoid unexpected behaviour, it would be best to stop all running containers before proceeding.

```
$ docker stop $(docker ps -q)
```

You should be good to continue now...

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.

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`.

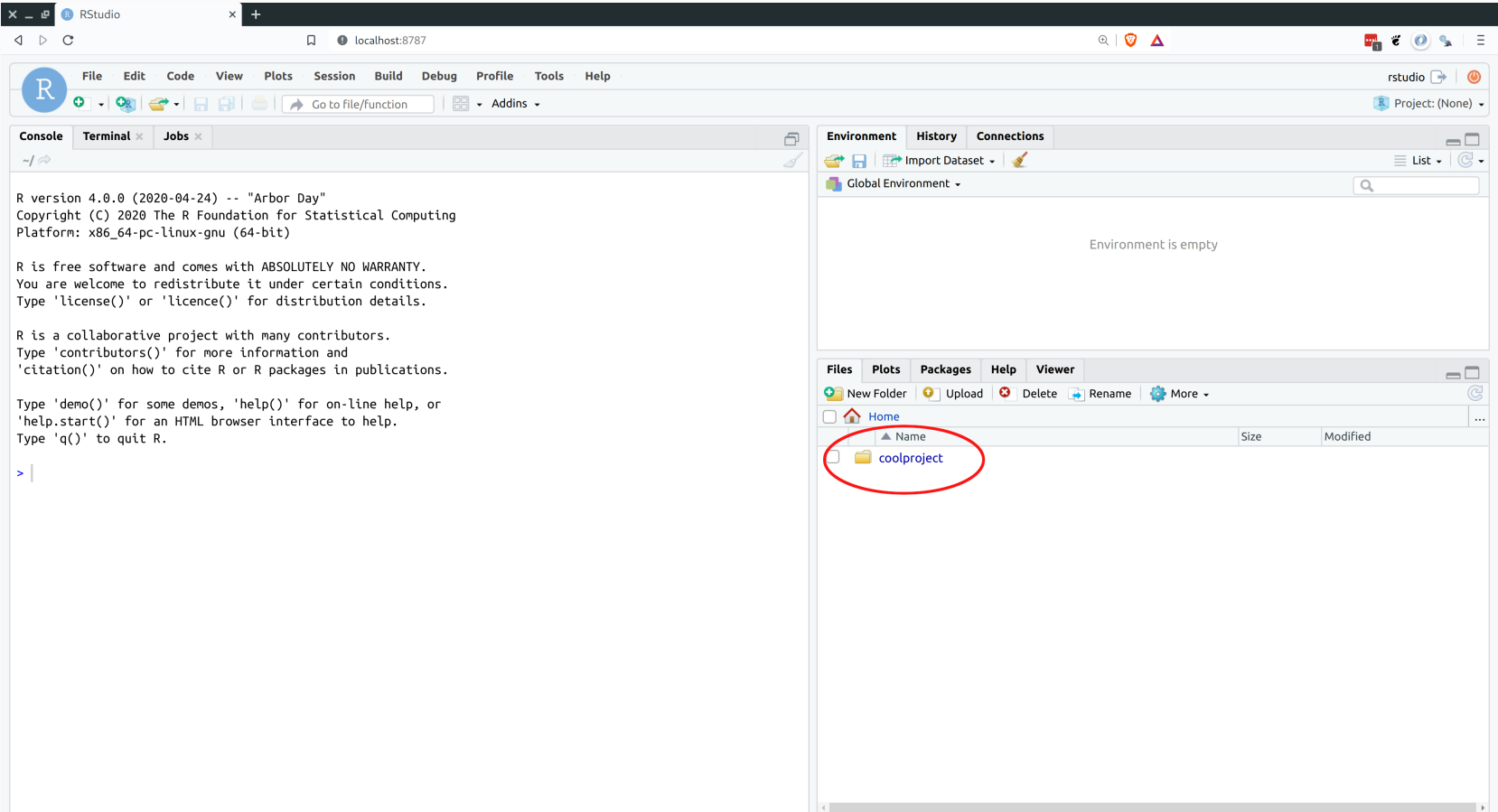
For example, say I have a folder on my computer located at `/home/grant/coolproject`. I can make this available to my "tidyverse" container by running:

```
$ 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 of 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.[†]

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.)

[†] 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.[†]

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.

[†] Exception: If you assigned a different default user than "rstudio" ([back here](#)).

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.

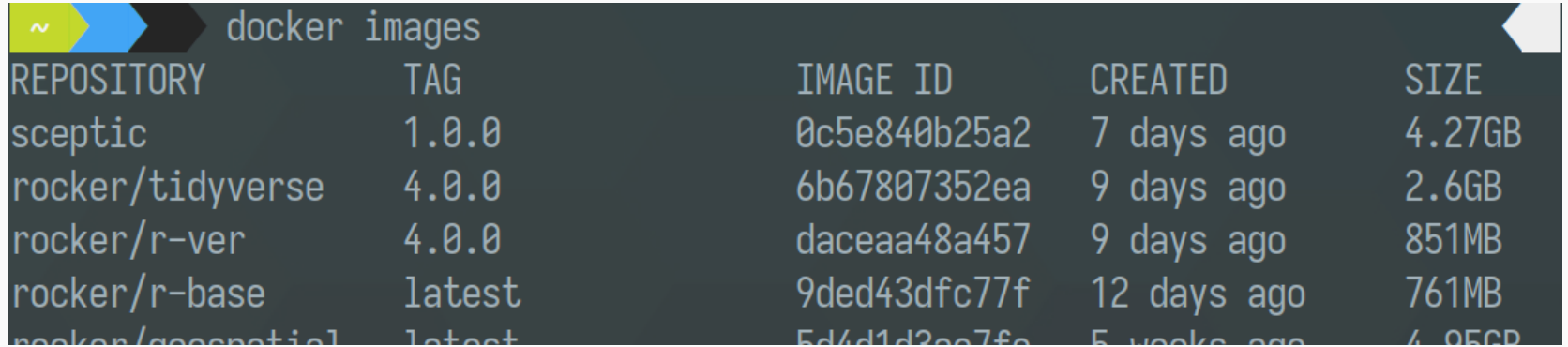
To see a list of the images¹ on your system, simply type:

```
$ docker images
```

¹ **Remember:** Images are distinct from containers.

Removing images

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

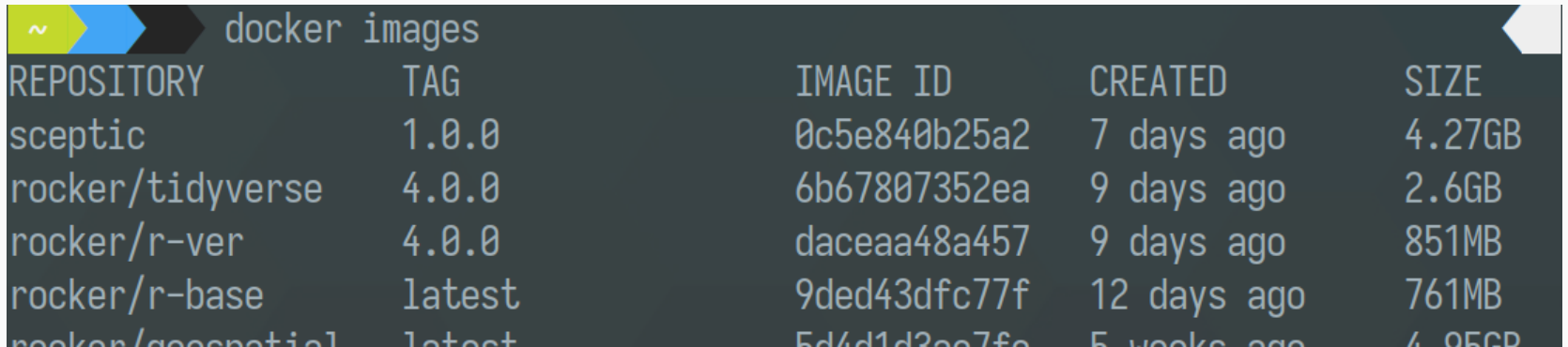


```
~ docker images
```

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/rosetta1	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 these dangling images 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 these dangling images 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 reproducibility, testing, and deployment.

Terminology analogy

- Dockerfile = sheet music
- Docker image = MP3 recording
- Container = MP3 being played on my 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
```

(See next slide for a list of key commands.)

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 <container-ids>` 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](#)
- [R Journal article \(Nüst et. al., 2020\)](#)
- [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.)

Table of contents

1. Prologue
2. Docker 101
3. Examples
 - Base R
 - R-dev
 - RStudio+
4. Write your own Dockerfiles & images
5. Sharing files with a container
6. Cleaning up
7. Conclusions