APIs: you're probably not using them and why you probably should

Presented by: **Josiah Parry**

RStudio

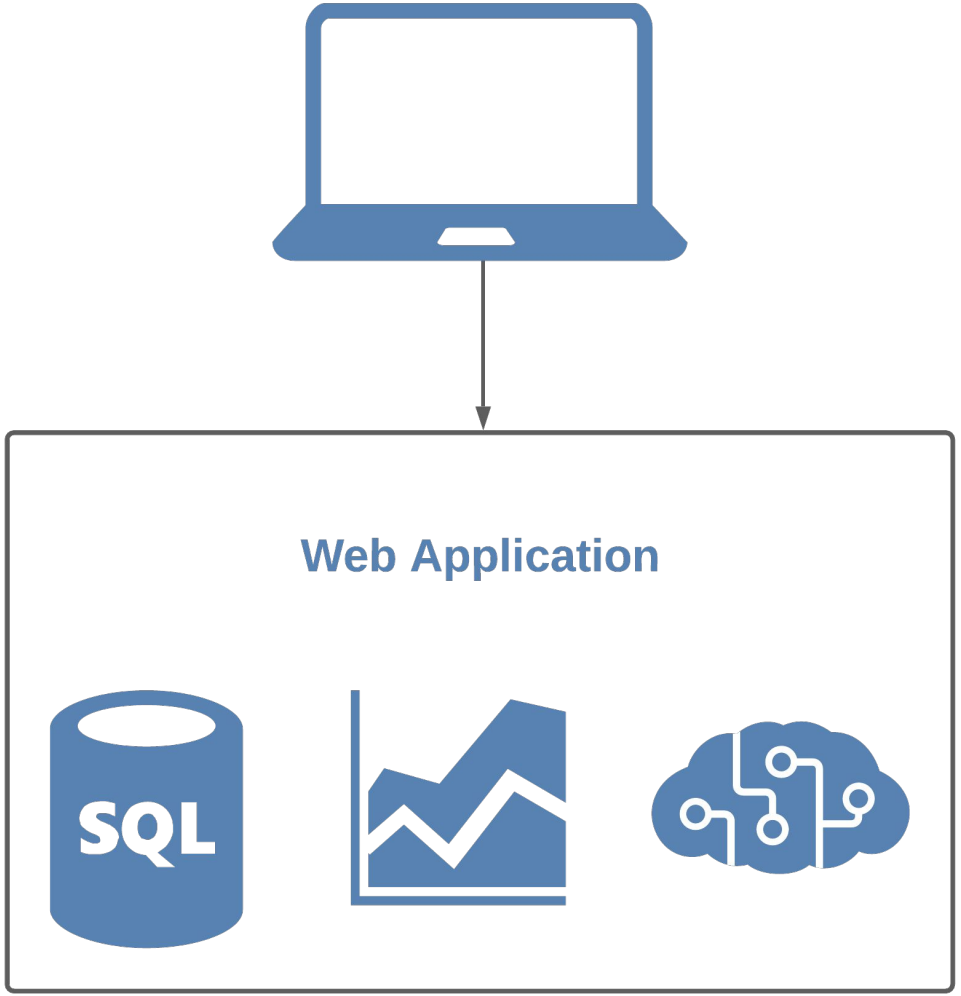
Changing Data Science Landscape

RStudio

Data Science in 2020

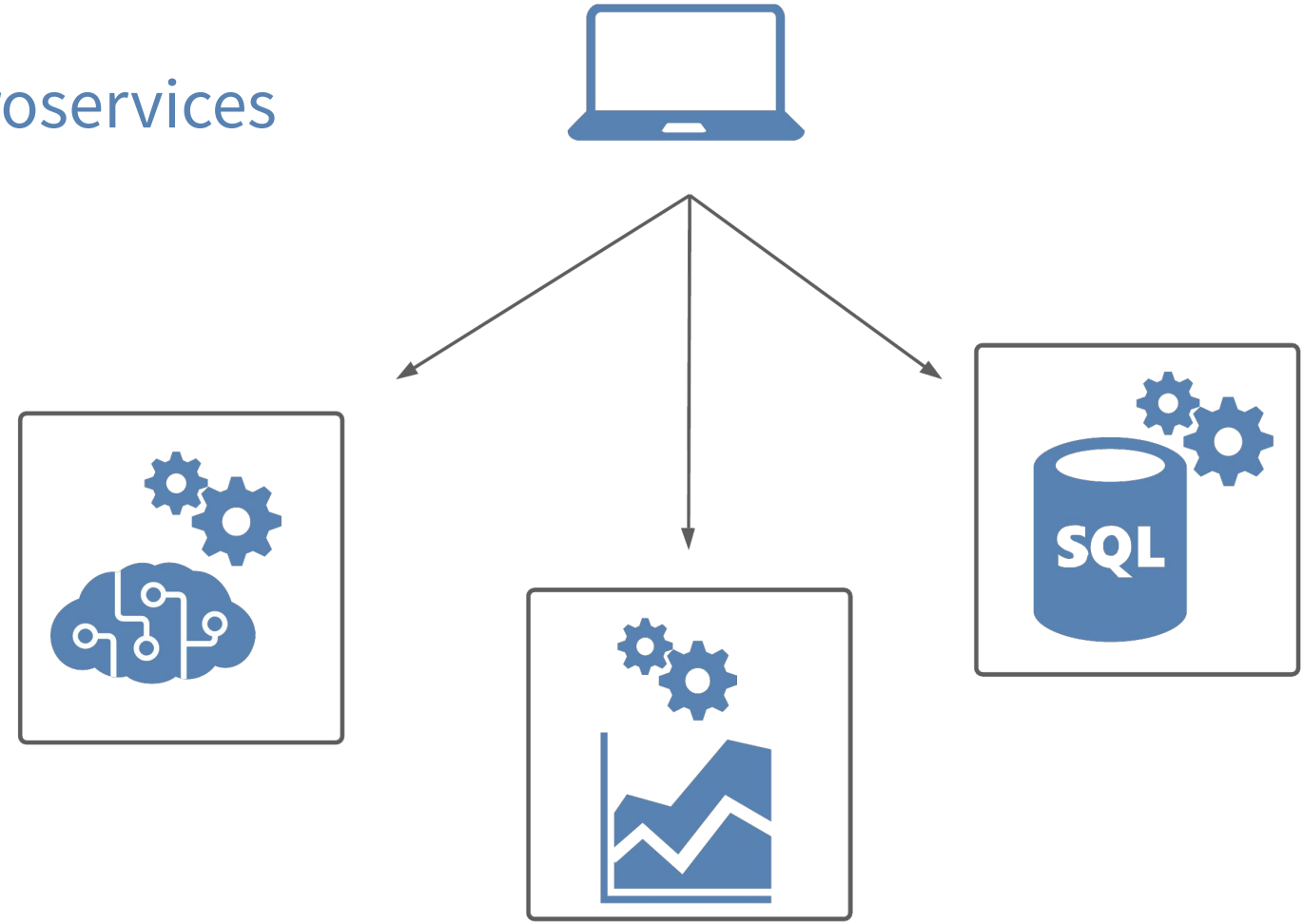
- Part of production applications
- Beyond ad hoc reports, visualizations, and dashboards
- Need to equip data scientists with production tools
- Dependence on IT
- To succeed, we need to embrace DevOps
- How? Microservices

Monolith



RStudio

Microservices



RStudio

“Gather together those things that change for the same reason, and separate those things that change for different reasons.”

Okay, but how?

APIs, that's how.

AP-who?

TL;DR - APIs

- Application Programming Interfaces
- Machines talking to machines
- RESTful APIs use HTTP
- Think of HTTP as a universal language

Conceptualizing APIs

- Think of an API like a function
- Functions have *arguments* and do something
- APIs have *parameters* and do something
- `function(argument = value)`
- `http://hostname:8888/api_endpoint/?parameter=value`

Data Science Tool Box

- PDF reports
- PowerPoint presentations
- Spreadsheets
- Interactive Dashboards
- *Etc.*
- Limited to end user consumption

RStudio

Data Science Tool Box

- PDF reports
- PowerPoint presentations
- Spreadsheets
- Interactive Dashboards
- *Etc.*
- **~ APIs ~**

RStudio

Why an API?

Organizational

- Smaller pieces are easier to manage (less code!)
- Component Monitoring
- Infrastructure for recycled code
  - Less technical debt!
- Better change management
  - Upgrade one service at a time without bringing down whole application

Collaborative

- Less friction in tool handoff
- Empower teams to use preferred tool chain
- Concerned about API requests not software language
- Reduced effort in tool hand off
  - No need to translate work
  - Not blocked by communication
- Programmatic use of services by other teams
  - Whereas dashboards require user intervention
- ***Glue together API requests! Pipelines, baby!***

Signs you might be ready to create an API

- Redundant code (copying and pasting/reusing)
- Other teams want your code
- Other teams may not know your language
- You want to integrate different software into one toolchain

How do you build an API?

API Frameworks

- R - plumber
- Python - Flask, Fast API
- JavaScript - Node.js & Express.js

How do you deploy an API?

Deployment Options

- RStudio Connect
- Digital Ocean
- Docker

DigitalOcean

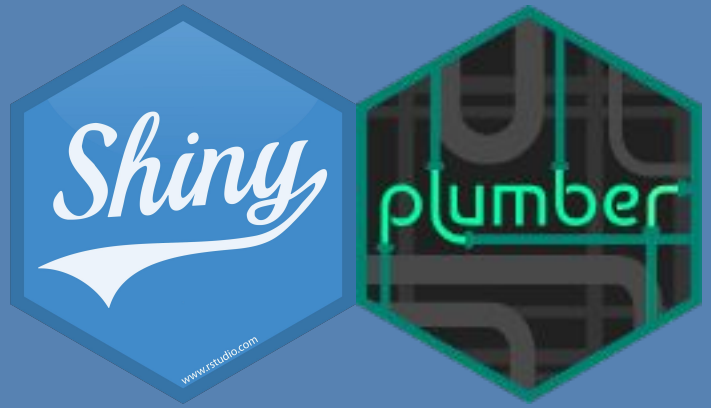
docker

RStudio

Why limit your stack to

RStudio

When it could be this



Let’s get into it

Parts of an API

- Host (fixed)
  - `http://api.hostname.io/`
- Endpoint
  - Resource location (think of as a function)
  - `http://api.hostname.io` **`/end-point`**
- Parameters (optional)
  - Address varying parts of a request
  - `http://api.hostname.io/end-point/` **`?param`**`=value`
- Headers & body (optional) (not in URL)
  - Associated (meta)data

API Requests

- Each API endpoint has a different **method**
- **GET**
  - Used to *retrieve* data. Parameters only. No body.
  - Everything is in the URL.
  - Don’t send sensitive data!
- POST
  - Used for *sending* data (files or text). More secure.
  - Creating or modifying something.
- Other methods:
  - PUT
  - DELETE (yikes!)
  - HEAD
  - …

APIs as a Maturity Process

- The one off analysis
- The *what ifs*?
  - Parameterized Rmd
  - Shiny Dashboard
- Need programmatic access
  - Function
  - API

Example: Global Crop Yields

- TidyTuesday Week 36, 2020
- Goal:
  - Fetch crop yields for a given crop, year, and country
- Ad hoc analysis
- Parameterized Rmd
- Function
- Plumber API

RStudio

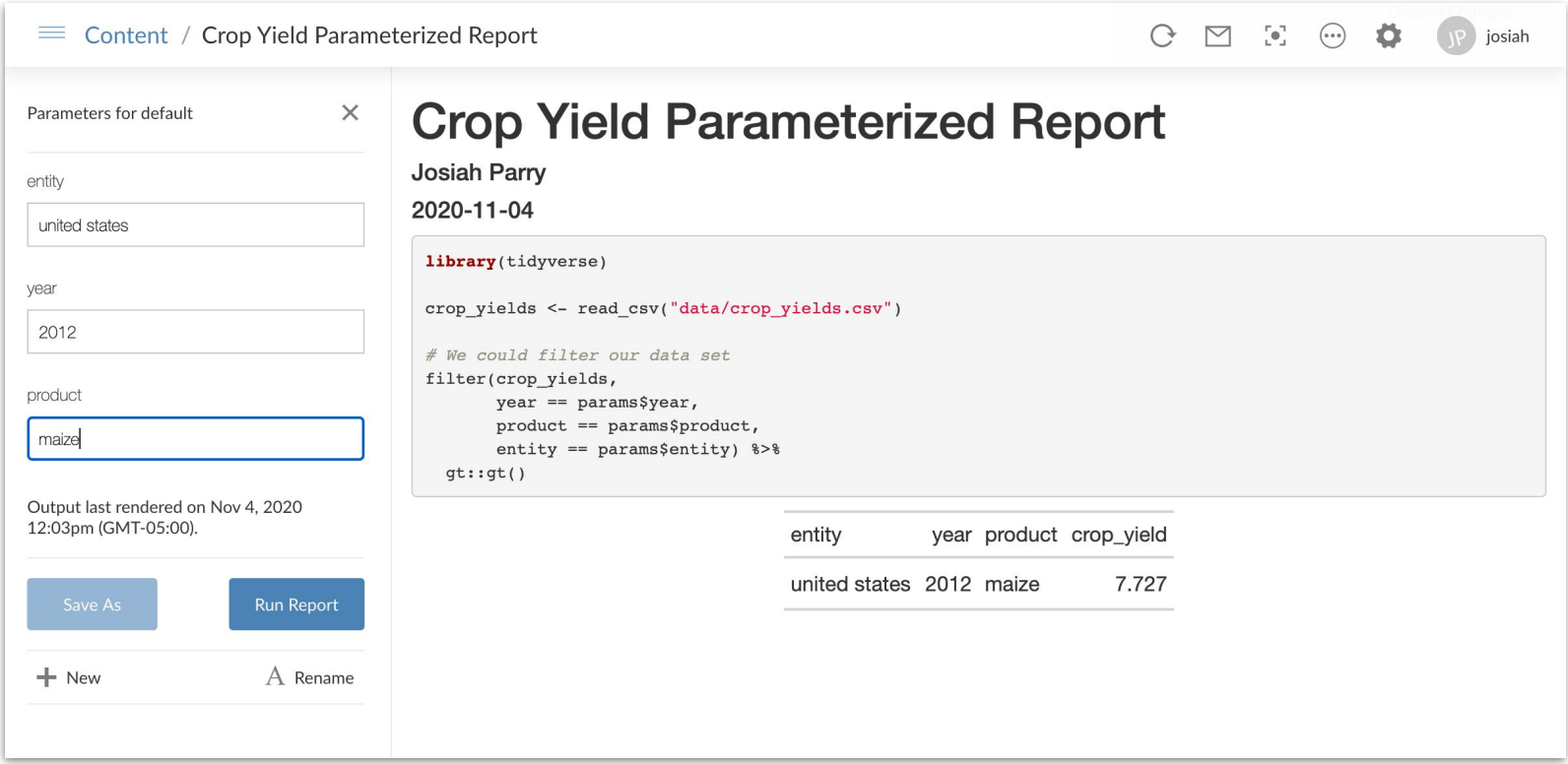
Example: ad hoc analysis

```
library(tidyverse)

crop_yields <- read_csv("data/crop_yields.csv")

# We could filter our data set
filter(crop_yields,
       year == 2018,
       product == "maize",
       entity == "cuba")
```

```
## # A tibble: 1 x 4
##   entity  year product crop_yield
##   <chr>  <dbl> <chr>        <dbl>
## 1 cuba    2018 maize         2.39
```

RStudio

Parameterized Rmd



RStudio

Making it functional

- Functions are code shortcuts!
- Functions are a special kind of object
- Arguments are placeholders
- Whatever is printed last is returned

```
my_fun <- function(arg) {
 # do something with the argument
 arg
}

my_fun("Hello!")
## [1] "Hello!"
```

RStudio

What is changing?

- Things that change should be arguments

```
filter(crop_yields,
       year == 2018,
       product == "maize",
       entity == "cuba")
```

- Making the skeleton

```
filter_yields <- function(.year, .product, .entity) {
    # This is where our filtering will go!
}
```

- Using a `.` to avoid confusion in the `filter()`

Example: making it functional

```
filter_yields <- function(.year, .product, .entity) {

 filter(crop_yields,
      year == .year,
      product == .product,
      entity == .entity)
}

filter_yields(2012, "potatoes", "guatemala")
```

```
## # A tibble: 1 x 4
##    entity     year product  crop_yield
##    <chr>     <dbl> <chr>         <dbl>
## 1 guatemala  2012 potatoes       25.0
```

You've basically made an API already


host
RSC content path
endpoint
https://colorado.rstudio.com/rsc/crop-yield-api/crop-yield
?.year=1978&.product=cassava&.entity=congo
parameters

The API

```
library(plumber)
library(tidyverse)

crop_yields <- read_csv("data/crop_yields.csv")

#* @apiTitle Crop Yields API
#* Return crop yields for a given product, year, and entity.
#* @param .year The year of interest (1961 - 2018)
#* @param .product The crop of interest. One of wheat, rice, maize, soybeans,
potatoes, beans, peas, cassava, barley, cocoa beans, or bananas.
#* @param .entity The country of interest.
#* @get /crop-yield

function(.year, .product, .entity) {
 filter(crop_yields,
        year == .year,
        product == .product,
        entity == .entity)
}
```

RStudio

The API

```
library(plumber)
library(tidyverse)

crop_yields <- read_csv("data/crop_yields.csv")

#* @apiTitle Crop Yields API
#* Return crop yields for a given product, year, and entity.
#* @param .year The year of interest (1961 - 2018)
#* @param .product The crop of interest. One of wheat, rice, maize, soybeans,
potatoes, beans, peas, cassava, barley, cocoa beans, or bananas.
#* @param .entity The country of interest.
#* @get /crop-yield

function(.year, .product, .entity) {
 filter(crop_yields,
        year == .year,
        product == .product,
        entity == .entity)
}
```

RStudio

The API

```
library(plumber)
library(tidyverse)

crop_yields <- read_csv("data/crop_yields.csv")

#* @apiTitle Crop Yields API
#* Return crop yields for a given product, year, and entity.
#* @param .year The year of interest (1961 - 2018)
#* @param .product The crop of interest. One of wheat, rice, maize, soybeans,
potatoes, beans, peas, cassava, barley, cocoa beans, or bananas.
#* @param .entity The country of interest.
#* @get /crop-yield

function(.year, .product, .entity) {
 filter(crop_yields,
        year == .year,
        product == .product,
        entity == .entity)
}
```

RStudio

The API

```
library(plumber)
library(tidyverse)

crop_yields <- read_csv("data/crop_yields.csv")

#* @apiTitle Crop Yields API
#* Return crop yields for a given product, year, and entity.
#* @param .year The year of interest (1961 - 2018)
#* @param .product The crop of interest. One of wheat, rice, maize, soybeans,
potatoes, beans, peas, cassava, barley, cocoa beans, or bananas.
#* @param .entity The country of interest.
#* @get /crop-yield

function(.year, .product, .entity) {
 filter(crop_yields,
        year == .year,
        product == .product,
        entity == .entity)
}
```

RStudio

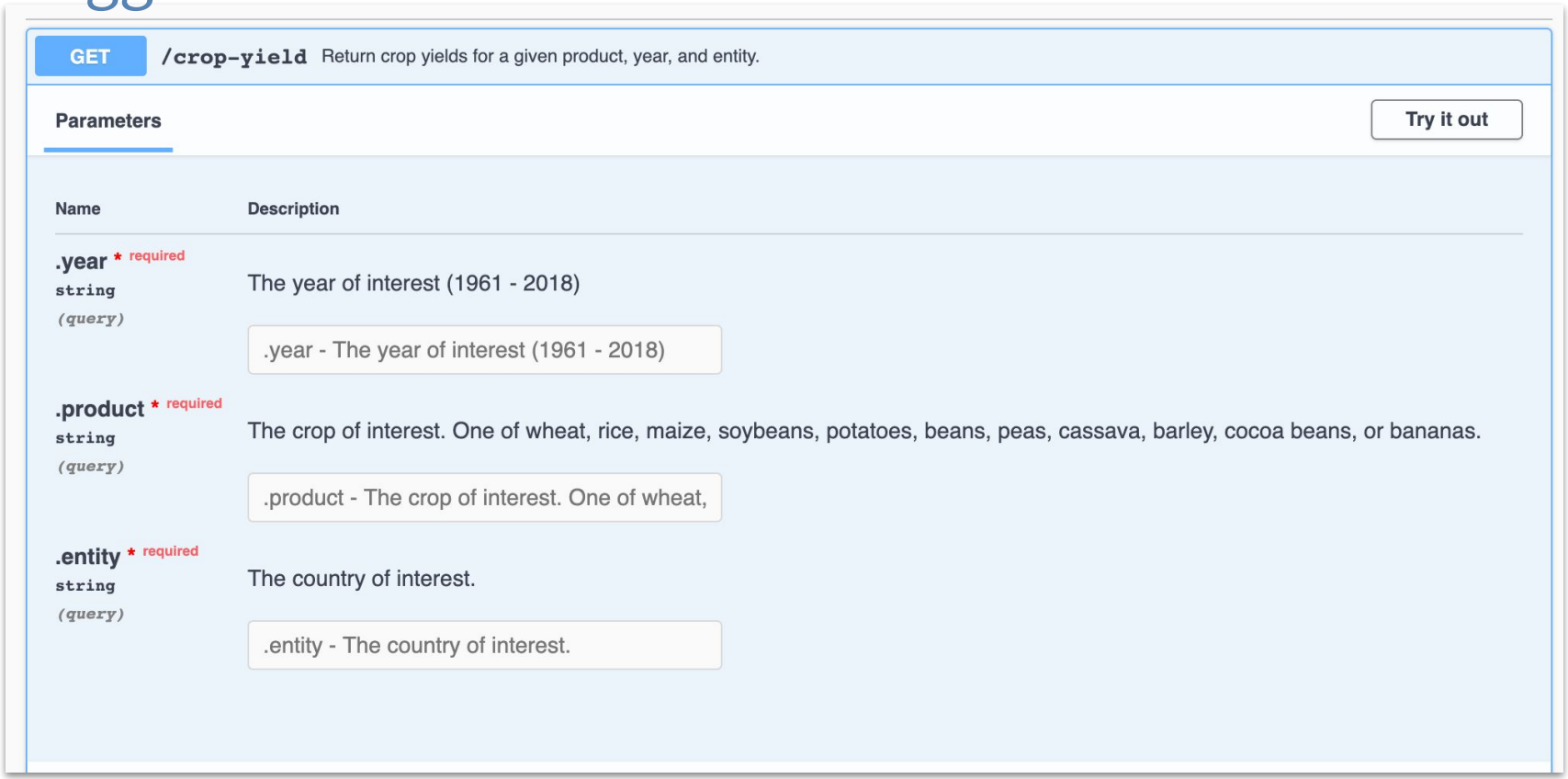
The API

```
library(plumber)
library(tidyverse)

crop_yields <- read_csv("data/crop_yields.csv")

#* @apiTitle Crop Yields API
#* Return crop yields for a given product, year, and entity.
#* @param .year The year of interest (1961 - 2018)
#* @param .product The crop of interest. One of wheat, rice, maize, soybeans,
potatoes, beans, peas, cassava, barley, cocoa beans, or bananas.
#* @param .entity The country of interest.
#* @get /crop-yield

function(.year, .product, .entity) {
 filter(crop_yields,
        year == .year,
        product == .product,
        entity == .entity)
}
```

RStudio

The Swagger Interface

Swagger



RStudio

Calling the API from R
{httr}

API Call Steps:

- This pattern is the same for all languages
- Generate the query URL
- Make the request
- Parse the response

Creating the URL

1. Specify the base URL
2. Create named list with parameters and values
3. Use `modify_url()` to fill in parameters

```
b_url <- "https://colorado.rstudio.com/rsc/crop-yield-api/crop-yield"

params <- list(.year = 1975, .product = "beans", .entity = "germany")

(query_url <- modify_url(url = b_url, query = params))
```

```
"https://colorado.rstudio.com/rsc/crop-yield-api/crop-yield?.year=1975&.product=beans&.entity=germany"
```

RStudio

Send the request

- Use the appropriate method
  - e.g. `GET()`, `POST()`, `PUT()`

```
(res <- GET(query_url))
```

```
## Response
```

[https://colorado.rstudio.com/rsc/crop-yield-api/crop-yield?.year=1975&.product=beans&.entity=germany]

```
##   Date: 2020-11-08 20:34
##   Status: 200
##   Content-Type: application/json
##   Size: 72 B
```

Parse the request

```
(resp_raw <- content(res, as = "text"))
```

```
"[{\"entity\":\"germany\",\"year\":1975,\"product\":\"beans\",\"crop_yiel
d\":2.8749}]"
```

```
jsonlite::fromJSON(resp_raw)
```

```
##    entity year product crop_yield
## 1 germany 1975   beans     2.8749
```

Recap

Recap

- APIs: computers talking to computers
- RESTful APIs use HTTP
- HTTP is language agnostic
- Add APIs to your DS toolbox
- Reduce communication barriers
- Enable programmatic use of tools

Thank you!

RStudio

Let’s talk about it.