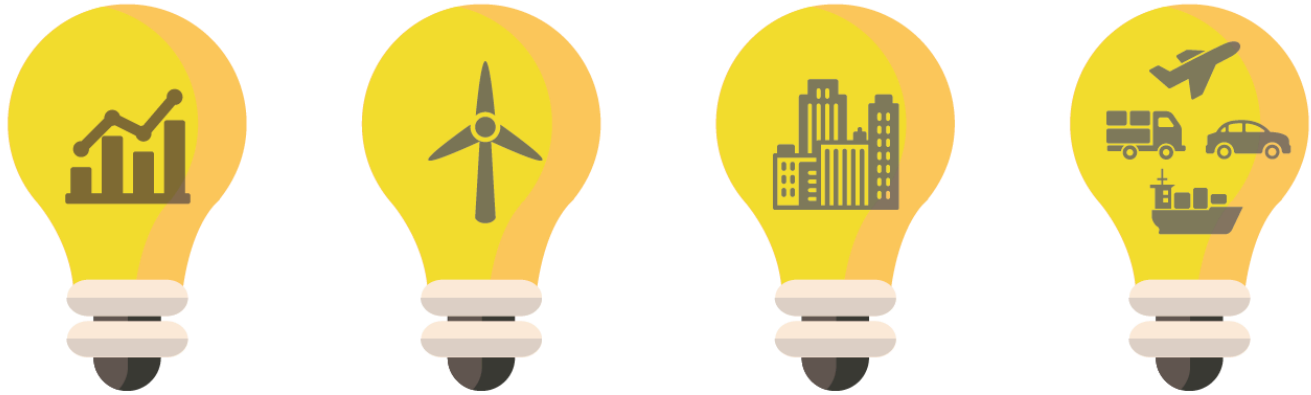
BENV0091 Lecture 8: APIs

Patrick de Mars



Lecture Overview

- String formatting
- APIs:
  - Introduction
  - National Grid Carbon Intensity
  - Met Office

String Formatting

Paste Functions

- The functions `paste()` and `paste0()` are used to join character vectors (strings)
  - `paste()`: arguments are separated by spaces (by default)
  - `paste0()`: arguments are not separated
- Task: use `paste()` to print:
  - ”The weather outside is {adjective}”
- Task: use `paste0()` to print:
  - ”The time is: {time}”

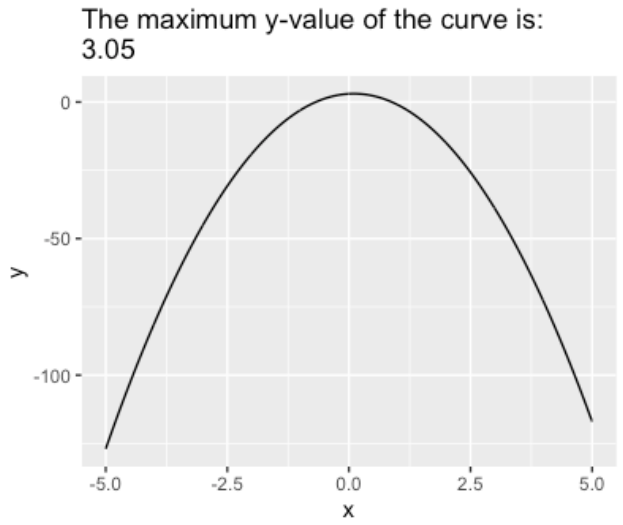
Use `now()` to return the current time

Using paste in a figure title

```
x <- seq(-5, 5, 0.01)
y <- -5*x**2 + x + 3
df <- tibble(x = x, y = y)

title <- paste0("The maximum y-value of the curve is: \n", max(y))

ggplot(df, aes(x = x, y = y)) +
  geom_line() +
  labs(title = title)
```



C-Style String Formatting

- It can be challenging or verbose to use paste functions to insert variables in the middle of a string:
  - “Your train departs at HH:MM and is expected to arrive at HH:MM”
  - ”The wind speed is {x}mph”

```
sprintf("The wind speed is: %smph", wind_speed)
```

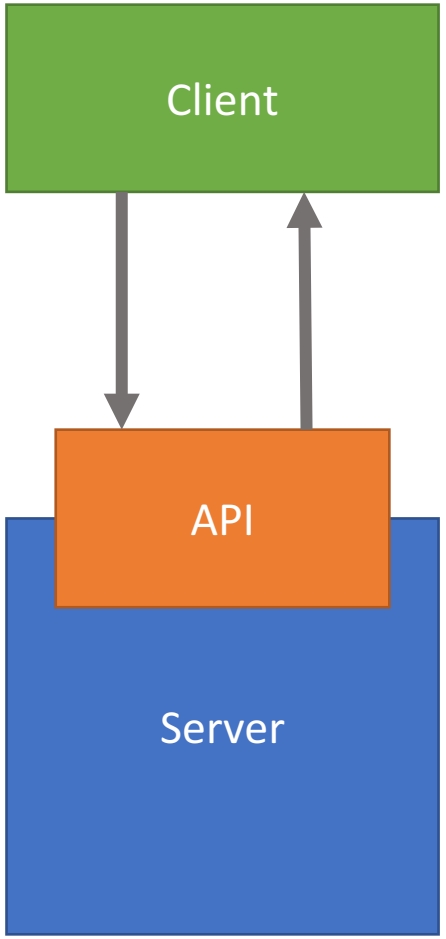
- The `sprintf()` function enables variables to be included in strings using C-style formatting
- Task: create a vector of normally distributed values using `rnorm(1000)` then print:
  - ”The mean of the data is {mean} and the standard deviation is {sd}”

```
vals <- rnorm(1000)
```

APIs

APIs

- An **Application Programming Interface** (API) is a set of rules or conventions determining how a user (**client**) can request **resources** from a **server**
- **Server**: runs the API and runs any server-side code
- **Client**: makes **requests** and exchanges data with the server via the API
- **API**: interprets requests to facilitate the exchange of data between client and server



API Examples
- Weather forecasts
- Flight times
- Social media updates

APIs in R

- Hypertext Transfer Protocol (HTTP) is a standard for transferring resources (data, results, code, webpages…) via the Web
- In R, the `httr` package is used for handling HTTP requests and responses
- There are a number of HTTP methods that can be used to specify requests, such as:
  - GET: to request a resource
  - POST: to create a resource
  - PUT: to change a resource
  - DELETE: to remove a resource
- In this lecture we will focus on GET
- Task: install the `httr` package

National Grid's Carbon Intensity API

- The National Grid's Carbon Intensity API is an open-access API that can be used to retrieve forecast and estimated Carbon Intensities for the GB power system
- Task: use GET to make a **request** for the current carbon intensity, assigning the **response** to a variable `r`
- Each API is unique and requires its own methods for extracting information from responses

```
r <- GET('https://api.carbonintensity.org.uk/intensity')
```

Requesting current carbon intensity

```
> names(r)
 [1] "url"         "status_code" "headers"     "all_headers" "cookies"     "content"
 [7] "date"        "times"       "request"     "handle"
```

Attributes of the response

API Response

- The HTTP response includes the following elements:
  - Status code
  - Headers
  - Content (body)
- The headers contain metadata about the response such as the response format and the file size
- Task: check the **status code** of your response with `r$status_code`
  - A successful request returns a status code of 200
  - 404 indicates a bad URL

API Response: Body

```
> content(r, 'text', encoding = 'UTF-8')
[1] "{ \r\n  \"data\":[{ \r\n    \"from\": \"2021-11-29T19:00
Z\",\r\n    \"to\": \"2021-11-29T19:30Z\",\r\n    \"intensity
\": {\r\n      \"forecast\": 194,\r\n      \"actual\": 204,\r
\n      \"index\": \"moderate\"\r\n    }\r\n  }]\r\n}"
```

- The body of an API response contains the information we have requested
- In this case the response is in **JSON format**
- The `content()` function can be used to retrieve the body of a response as **plain text (**or in a raw format)
- However, the text is still not readable in this form: the `prettify()` function (jsonlite) can be used to render the plain JSON text to a readable format
- This is readable but still not a convenient R object to handle
- Task: use `prettify()` to print the JSON response (see right)

prettify(content(r, 'text'))


```
{
    "data": [
        {
            "from": "2021-11-02T15:00Z",
            "to": "2021-11-02T15:30Z",
            "intensity": {
                "forecast": 248,
                "actual": 262,
                "index": "high"
            }
        }
    ]
}
```

JSON

- JSON stands for JavaScript Object Notation and is a file format, consisting of key-value pairs
- JSON supports **nested data** such as in the example on the right
  - Nested data cannot be represented in CSV files!
- JSON files are handled with the `jsonlite` package in R
- Task: install `jsonlite`

Example JSON


```
{
    "team1": {
        "Rafael": {
            "age": 35,
            "hand": "left"
        },
        "Roger": {
            "age": 40,
            "hand": "right"
        }
    },
    "team2": {
        "Novak": {
            "age": 34,
            "hand": "right"
        },
        "Andy": {
            "age": 34,
            "hand": "right"
        }
    }
}
```

Response Formats: JSON, XML, CSV

- APIs can respond with any data format so long as it can be serialised
- Most APIs use XML and/or JSON as the response format for data
- *The BMRS API is a notable exception: it returns XML or CSV*

JSON

- Easy to handle with a JSON parser (e.g. jsonlite in R)
- Not as flexible (e.g. does not support metadata)

XML

- Long history and well-known by developers; supports browser rendering
- Difficult to interpret for non-developers

CSV

- Easy to interpret for data scientists
- Does not support hierarchical data

Convert from JSON to an R Object

```
r <- GET('https://api.carbonintensity.org.uk/intensity')
parsed <- fromJSON(content(r, 'text'))
df <- parsed$data %>% unnest(cols = intensity)
```

- The `toJSON()` and `fromJSON()` functions (**jsonlite**) can be used to convert between JSON format and R objects such as data frames or lists
- Task:
  - use `fromJSON()` to convert the response to an R object (in this case a list)
  - Retrieve `data` from the parsed list
  - Finally, use `unnest()` (tidyr) to convert the list of data frames to regular columns
- *As you can see, there are necessarily bespoke solutions to parsing a API responses*
- Task: wrap the code above in a function that can be re-used

Nested data frame (in JSON format)

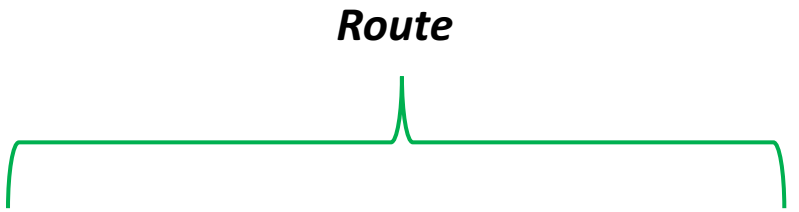

```
[
  {
    "from": "2021-11-29T20:30Z",
    "to": "2021-11-29T21:00Z",
    "intensity": {
      "forecast": 188,
      "actual": 182,
      "index": "moderate"
    }
  }
]
```


Unnested data frame (in JSON format)


```
[
  {
    "from": "2021-11-29T20:30Z",
    "to": "2021-11-29T21:00Z",
    "forecast": 188,
    "actual": 182,
    "index": "moderate"
  }
]
```

Routing and Endpoints

- Endpoints are resources which are available through the API such as:
  - Creating a plot
  - Returning data for a specific date
- Routes determine which endpoint the client is requesting, and is specified in the URL
- Each route can have multiple endpoints corresponding to different verbs: GET, POST etc.




```
GET('https://api.carbonintensity.org.uk/intensity/2020-01-01/2020-01-08')
```

This endpoint returns the carbon intensity between 2020-01-01 and 2020-01-08

More National Grid Functionality

- The Carbon Intensity API documentation describes endpoints for:
  - Regional carbon intensity
  - National generation mix
- Tasks:
  - Write a function to request the CO2 intensity data between two dates and parse it into a data frame
    - Write a function to calculate the **average** carbon intensity for the past week
    - Write a function to calculate the period of **highest** carbon intensity in the past week
  - Which regions have the lowest and highest carbon intensities right now?
  - Get the forecast carbon intensity for the next 24 hours (fw24h endpoints)

Datetimes must be in ISO8601 format YYYY-MM-DDThh:mmZ e.g. 2017-08-25T12:35Z

Remember the `sprintf()` and paste functions for string formatting

The `days()` function (lubridate) can be used to add or subtract days from datetimes

API Keys

- The National Grid API can be used without authentication
- Many APIs require an authentication key (or **access token**) to make a request
- Note: be careful not to publish API keys on Github! It is safer to store API keys on your local machine (for instance as **environment variables**) and retrieve them when needed
  - Environment variables should be stored in a special file: `.Renviron`
  - See `Sys.getenv()` for retrieving environment variables
  - Useful blog post on storing API keys as environment variables

Met Office API

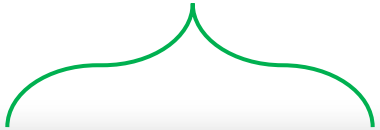
- The Met Office API can be used to retrieve location-specific or national forecasts
  - See documentation
- You should have already signed up to the Met Office DataPoint – your API Key is available on your account page.
- Task:
  - Use your API key and the URL below to request the site list
  - Parse the response to a data frame
  - Find the location ID for ”Land’s End”

```
# Getting site list
url <- sprintf('http://datapoint.metoffice.gov.uk/public/data/val/wxfcs/all/json/sitelist?key=%s', API_KEY)
```

Query Parameters

- So far we have seen **hierarchical parameters** (such as the date or location) separated by `/`
- However, **query (or optional) parameters** are used by many APIs for routing to endpoints
- Query parameters are separated from the hierarchical parameters by `?`, and separated from *each other* by `&`
- The code below has 1 hierarchical parameter (location ID) and 2 query parameters (resolution and API key)
- Task: use the URL below with your API key and the ID for Land's End to request the 3hourly forecast for Land's End

Query parameters

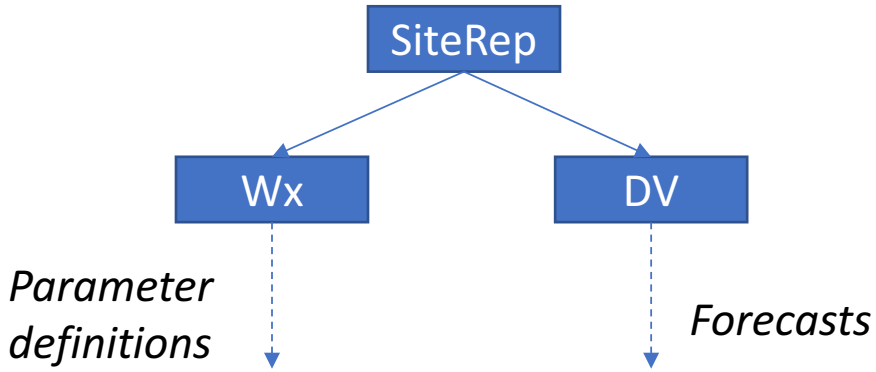


```
url <- sprintf('http://datapoint.metoffice.gov.uk/public/data/val/wxfcs/all/json/%s?res=3hourly&key=%s', location_id, API_KEY)
```

Retrieving the Forecast

- The request in the previous slide returns a 5-day forecast at 3-hourly resolution
- Task: run `prettify(content(r, 'text'))` to expose the structure of the JSON response
- You should find there are two levels after SiteRep:
  - Wx: defines the parameters
  - DV: contains location data and the forecast

JSON response structure (top levels)



Parameter definitions

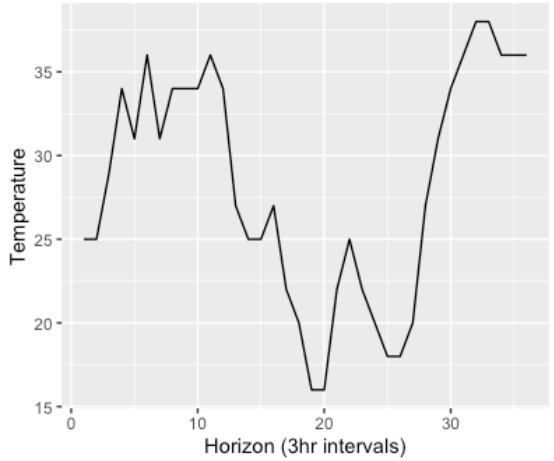
```
> data$SiteRep$Wx$Param
   name   units                          $
1     F       C    Feels Like Temperature
2     G     mph                Wind Gust
3     H       %  Screen Relative Humidity
4     T       C              Temperature
5     V                        Visibility
6     D compass           Wind Direction
7     S     mph               Wind Speed
8     U                    Max UV Index
9     W                    Weather Type
10   Pp       % Precipitation Probability
```

Retrieving the Forecast

- The data is buried even deeper within the response
- The Rep object is a list of 5 data frames with the forecast for the following 5 days
- Task: use a **for loop** to create a vector giving the Wind Speed for the next 5 days (at 3 hourly intervals)

```
> data$SiteRep$DV$Location$Period$Rep
[[1]]
    D F  G  H Pp  S  T  V  W U    $
1   W 7 34 99 19 25 11 MO  8 1  720
2   W 7 34 99 11 25 11 MO  7 1  900
3 WSW 7 38 91 74 29 11 VG 12 0 1080
4 WSW 7 45 96 98 34 12 GO 15 0 1260

[[2]]
    D F  G  H Pp  S  T  V  W U    $
1 WNW 7 40 90 91 31 12 EX 12 0    0
2  NW 6 47 84 36 36 11 EX  7 0  180
3  NW 5 43 75  8 31 10 EX  7 0  360
4  NW 5 43 73 22 34 10 EX  7 1  540
5  NW 4 43 78 43 34  9 EX 10 1  720
6  NW 4 47 71 45 34  9 EX  7 1  900
7 NNW 4 47 72 49 36  9 EX  7 0 1080
8   N 2 45 82 46 34  8 VG  9 0 1260
```



Period-Specific Forecasts

- By adding an addition query parameter `time`, it is possible to retrieve a forecast for a single period
  - Note: forecasts are only available at 3 hourly intervals: 0h, 3h, 6h,…,21h!
  - Note: the time must be in ISO 8601 format
- Task: obtain the forecast Feels Like Temperature for **tomorrow at midday**
- Task: wrap your code in a function with an argument specifying a time to retrieve the forecast

A lubridate solution to 24 hours ahead in ISO 8601 format

```
time <- (now() + hours(24)) %>% format_ISO8601()
```

```
forecast_tomorrow_feels_like_temp <- function(hour){
  # Your code goes here...
  return(feels_like_temp)
}
```

Some Useful APIs

- Elexon BMRS: docs (pdf)
  - Detailed GB electricity consumption and production data
- Renewables Ninja: docs
  - Simulated wind and solar PV generation
- Octopus Energy: docs
  - Tariff charges, electricity/gas smart meter reads etc.
- Open Weather: docs
  - Current, historical and forecast weather data
- Twitter: docs
  - Retrieve tweets, create tweets etc.
- European Environment Agency: docs
  - Environmental indicators, water quality etc.
- UK public sector APIs: catalogue
  - Companies House, NHS 999 callouts etc.
- OECD APIs: catalogue
  - Economic indicators

Building an API in R

- The **plumber** package can be used to build your own web API in R (a great way to improve your understanding of APIs!)
- You can test this out by running it **locally** – making it publicly available can be achieved on RStudio Connect (at a cost)