13. Download and Unzip

Programmatic file downloads, archives, and HTTP requests

Dr. Paul Schmidt

To install and load all packages used in this chapter, run the following code:

```
for (pkg in c("tidyverse", "httr2", "withr")) {
  if (!require(pkg, character.only = TRUE)) install.packages(pkg)
}

library(tidyverse)
library(httr2)
```

Introduction

In many data analysis projects, the first step is obtaining data from an external source — a public repository, a government data portal, or a colleague's shared folder. The most common approach is to download the file manually via a web browser and place it in the project folder. While this works, it creates a gap in the reproducibility chain: anyone who wants to rerun the analysis must know where the file came from and how to get it.

Downloading files programmatically — directly from within an R script — closes this gap. The URL is documented in the code itself, making the data source explicit and traceable. If the source data is updated, rerunning the script fetches the latest version automatically. For workflows that involve many files or need to run on a schedule (e.g., weekly reports), manual downloading quickly becomes impractical, while a scripted approach scales effortlessly.

Beyond pure downloads, R can also interact with web APIs to query databases, retrieve weather data, or access any service that provides a programmatic interface. This chapter covers the essential tools for all of these tasks.

download.file()

The simplest way to download a file in R is `download.file()`. It takes a URL and a destination path, and saves the file to disk:

```
url <- "https://raw.githubusercontent.com/allisonhorst/palmerpenguins/main/inst/
extdata/penguins.csv"
dest <- "penguins.csv"

download.file(url, destfile = dest)
```

After running this code, the file `penguins.csv` appears in the working directory. We could then read it with `read_csv("penguins.csv")` as usual.

Binary Mode on Windows

One critical detail when working on Windows is the `mode` argument. By default, `download.file()` uses text mode, which can corrupt binary files like ZIP archives, Excel spreadsheets, or images. To be safe, always use `mode = "wb"` (write binary) for non-text files:

```
# For ZIP, Excel, images, PDFs — always use mode = "wb"
download.file(
  url = "https://example.com/data.zip",
  destfile = "data.zip",
  mode = "wb"
)
```

For plain text files like CSV or TSV, the default mode works fine. When in doubt, `mode = "wb"` never hurts — it works correctly for text files too.

Timeout for Large Files

R's default download timeout is 60 seconds. For large files or slow connections, this may not be enough. The timeout can be increased via `options()`:

```
# Increase timeout to 5 minutes (300 seconds)
options(timeout = 300)

download.file(url, destfile = "large_file.zip", mode = "wb")
```

It is good practice to set the timeout at the top of a script if large downloads are expected, rather than discovering the issue mid-execution.

HTTPS and Download Methods

On some systems, `download.file()` may fail with HTTPS URLs. Specifying `method = "libcurl"` usually resolves this:

```
download.file(url, destfile = dest, method = "libcurl")
```

On modern R installations (4.2+), `"libcurl"` is typically the default, so this is mainly relevant for older setups or restricted environments.

Exercise: Download a CSV and Read It

The Palmer Penguins dataset is available as a CSV file at:

```
https://raw.githubusercontent.com/allisonhorst/palmerpenguins/main/inst/extdata/penguins.csv
```

1. Download the file to a temporary location using `tempfile(fileext = ".csv")` as the destination.
2. Read the downloaded file with `read_csv()`.
3. Use `glimpse()` to inspect the result.

Solution

```
url <- "https://raw.githubusercontent.com/allisonhorst/palmerpenguins/main/inst/extdata/penguins.csv"
tmp <- tempfile(fileext = ".csv")

download.file(url, destfile = tmp)

penguins <- read_csv(tmp)
glimpse(penguins)
```

The `glimpse()` output would show 344 rows and 8 columns including species, island, bill measurements, flipper length, body mass, sex, and year. Using `tempfile()` ensures the download does not clutter the working directory — more on this in the next section.

Working with Temporary Files

When downloading data that only needs to exist for the duration of a script, there is no reason to save it permanently. R provides built-in functions for creating temporary files and directories that are automatically cleaned up when the R session ends.

tempfile() and tempdir()

The function `tempfile()` generates a unique file path in R's temporary directory. The file does not exist yet — `tempfile()` merely creates a name that is guaranteed not to collide with existing files:

```
# Each call generates a unique path
tempfile()
```

```
[1] "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpQxbAPk\\fileab403ac26b3f"
```

```
tempfile()
```

```
[1] "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpQxbAPk\\fileab4046cc2705"
```

```
# Specify a file extension
tempfile(fileext = ".csv")
```

```
[1] "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpQxbAPk\\fileab403fee51e7.csv"
```

```
# Specify a pattern (prefix) for readability
tempfile(pattern = "penguins_", fileext = ".csv")
```

```
[1] "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpQxbAPk\
\penguins_ab40298d4eda.csv"
```

The temporary directory itself can be inspected with `tempdir()`:

```
tempdir()
```

```
[1] "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpQxbAPk"
```

This directory is session-specific. When R exits, the operating system eventually cleans it up. This means temporary files are a good default for intermediate downloads — they keep the working directory clean and do not accumulate over time.

withr::local_tempfile() for Automatic Cleanup

The {withr} package provides `local_tempfile()`, which creates a temporary file that is automatically deleted when the enclosing function or code block finishes. The temporary file is automatically deleted when the enclosing function returns, even if an error occurs:

```
library(withr)

process_remote_data <- function(url) {
  tmp <- local_tempfile(fileext = ".csv")
  download.file(url, destfile = tmp)

  data <- read_csv(tmp)
  # tmp is automatically deleted when this function returns
```

```
  data
}
```

For interactive use, `tempfile()` is perfectly adequate. The `local_tempfile()` approach becomes valuable in functions and packages where you want to guarantee cleanup even if an error occurs.

When to Use Temporary vs. Permanent Files

As a rule of thumb: use temporary files when the raw download is merely an intermediate step (e.g., downloading a ZIP only to extract its contents). Use permanent files when the download itself is the result, or when you want to cache the data to avoid repeated downloads.

Unzipping Archives

Many data sources distribute files as ZIP archives. R can handle these directly without external tools.

Inspecting a ZIP File

Before extracting, it is often useful to see what a ZIP archive contains. The `unzip()` function with `list = TRUE` returns a data frame of the archive's contents:

```
# Create a small ZIP for demonstration
zip_path <- tempfile(fileext = ".zip")
csv1 <- tempfile(fileext = ".csv")
csv2 <- tempfile(fileext = ".csv")
write.csv(mtcars[1:5, ], csv1, row.names = FALSE)
write.csv(iris[1:5, ], csv2, row.names = FALSE)
zip(zip_path, files = c(csv1, csv2), flags = "-j") # -j: no directory paths

# List contents without extracting
unzip(zip_path, list = TRUE)
```

```
                  Name Length                Date
1 fileab4021dc218c.csv    263 2026-03-12 10:36:00
2 fileab4027502879.csv    195 2026-03-12 10:36:00
```

The `list = TRUE` argument returns a data frame of the archive's contents — file names, uncompressed sizes, and dates. It is a good practice to inspect before extracting, especially with archives from unfamiliar sources.

Extracting Files

To extract all files, call `unzip()` with the `exdir` argument specifying where to place the extracted files:

```
# Extract to a temporary directory
extract_dir <- tempdir()
unzip(zip_path, exdir = extract_dir)

# List extracted files
list.files(extract_dir, pattern = "\\.csv$")
```

To extract only specific files, use the `files` argument:

```
# Extract only one specific file
unzip(zip_path, files = "data.csv", exdir = extract_dir)
```

Reading Directly from a ZIP with unz()

For the common case where a ZIP contains a single CSV file, R provides `unz()` which creates a connection to a file inside a ZIP without extracting to disk:

```
# Read a CSV directly from inside a ZIP
data <- read_csv(unz(zip_path, "data.csv"))
```

This is elegant and efficient — no intermediate files, no cleanup needed. It works well with `read_csv()`, `read_tsv()`, `read.csv()`, and similar functions that accept connections.

Exercise: Create, Inspect, and Extract a ZIP

1. Create two small CSV files from built-in datasets (`PlantGrowth` and `ToothGrowth`) and bundle them into a single ZIP archive using `zip()` .
2. Inspect the archive contents with `unzip(..., list = TRUE)` .
3. Extract the files to a temporary directory.
4. Read one of the extracted CSV files with `read_csv()` and display the first few rows with `head()` .

Solution

```r
# Create CSV files and bundle into ZIP
csv_pg <- tempfile(fileext = ".csv")
csv_tg <- tempfile(fileext = ".csv")
write.csv(PlantGrowth, csv_pg, row.names = FALSE)
write.csv(ToothGrowth, csv_tg, row.names = FALSE)

zip_path <- tempfile(fileext = ".zip")
zip(zip_path, files = c(csv_pg, csv_tg), flags = "-j")

# Inspect
unzip(zip_path, list = TRUE)
```

```
                  Name Length                Date
1 fileab40517123af.csv    405 2026-03-12 10:36:00
2 fileab407e205cc1.csv    821 2026-03-12 10:36:00
```

```r
# Extract
extract_dir <- tempfile() # use tempfile() as unique dir name
dir.create(extract_dir)
unzip(zip_path, exdir = extract_dir)

# Read one of the extracted files
csv_files <- list.files(extract_dir, pattern = "\\.csv$", full.names = TRUE)
data <- read_csv(csv_files[1])
```

```
Rows: 30 Columns: 2
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr (1): group
dbl (1): weight

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
head(data)
```

```
# A tibble: 6 × 2
  weight group
   <dbl> <chr>
1   4.17 ctrl
2   5.58 ctrl
3   5.18 ctrl
4   6.11 ctrl
5   4.5  ctrl
6   4.61 ctrl
```

Using `tempfile()` as the extraction directory (instead of `tempdir()`) gives a clean, dedicated folder without leftover files from other operations.

httr2 for HTTP Requests

While `download.file()` is sufficient for straightforward downloads, many modern data sources provide their data through web APIs (Application Programming Interfaces). APIs typically return data in JSON format and may require specific headers, query parameters, or authentication. The {httr2} package provides a clean, pipe-friendly interface for building and executing HTTP requests.

The Request-Perform-Parse Pattern

The core workflow in {httr2} follows three steps: build the request, perform it, and parse the response:

```
library(httr2)

resp <- request("https://api.open-meteo.com/v1/forecast") %>%
  req_url_query(
    latitude = 52.52,
    longitude = 13.41,
    current_weather = "true"
  ) %>%
  req_perform()

resp
```

The `request()` function creates a request object from a base URL. The `req_url_query()` function appends query parameters (the part after `?` in a URL). Finally, `req_perform()` sends the request and returns a response object.

Inspecting the Response

The response object contains the HTTP status code, headers, and body. Several helper functions extract these pieces:

```
# HTTP status (200 = success)
resp_status(resp)

# Response body as a character string
resp_body_string(resp)

# Response body parsed from JSON into an R list
weather <- resp_body_json(resp)
weather$current_weather$temperature
```

For the Open-Meteo example above, `resp_body_json()` returns a nested list. The current temperature for Berlin would be accessible at `weather$current_weather$temperature`.

Adding Headers

Some APIs require custom headers, for example an API key or a specific content type. These are added with `req_headers()`:

```
request("https://api.example.com/data") %>%
  req_headers(
    Authorization = "Bearer YOUR_API_KEY",
    Accept = "application/json"
```

```
  ) %>%
  req_perform()
```

> ! Important
>
> Never hardcode API keys in scripts that are committed to version control. Instead, store them in environment variables and access them with `Sys.getenv("MY_API_KEY")`. This keeps credentials out of your Git history.

Comparison with download.file()

For simple file downloads, `download.file()` remains the most straightforward choice. Use {httr2} when you need any of the following: query parameters, custom headers, authentication, JSON parsing, error handling with retries, or when interacting with REST APIs. The two approaches are complementary rather than competing.

Practical Example - Weather Data from Open-Meteo

To illustrate a full API workflow, we query hourly temperature data from the Open-Meteo API. This API is free, requires no API key, and provides weather forecasts and historical data for any location on Earth.

Building the Request

The Open-Meteo API expects latitude, longitude, and a specification of which weather variables to return. We request hourly temperature for Berlin over the next 7 days:

```
resp <- request("https://api.open-meteo.com/v1/forecast") %>%
  req_url_query(
    latitude = 52.52,
    longitude = 13.41,
    hourly = "temperature_2m",
    timezone = "Europe/Berlin"
  ) %>%
  req_perform()
```

Parsing JSON into a Tibble

The JSON response contains nested lists. We extract the relevant parts and assemble them into a tidy tibble:

```
weather_data <- resp_body_json(resp)

forecast <- tibble(
  time = weather_data$hourly$time %>% unlist() %>% ymd_hm(),
  temperature = weather_data$hourly$temperature_2m %>% unlist()
)

forecast
```

The `time` field arrives as a character vector in ISO 8601 format, which `lubridate::ymd_hm()` (loaded with {tidyverse}) parses into proper datetime objects. The result is a clean tibble with one row per hour.

Visualizing the Forecast

With the data in tidy format, creating a plot is straightforward:

```
ggplot(forecast, aes(x = time, y = temperature)) +
  geom_line(color = "#00923f", linewidth = 0.6) +
  labs(
    title = "7-Day Temperature Forecast for Berlin",
    x = NULL,
    y = "Temperature (°C)"
  ) +
  theme_minimal()
```

This would produce a line chart showing the hourly temperature over the coming week. The diurnal cycle (warm during the day, cool at night) is typically clearly visible.

Exercise: Query Weather for a Different City

Use the Open-Meteo API to retrieve a 7-day hourly temperature forecast for a city of your choice.

1. Look up the latitude and longitude of your city (e.g., via Google Maps or latlong.net).
2. Modify the API request from the example above with the new coordinates.
3. Parse the response into a tibble and create a line plot of the temperature.
4. Bonus: add `hourly = "temperature_2m,precipitation"` to the query and plot both variables.

Solution

Here is an example for Vienna (latitude 48.21, longitude 16.37):

```
# Query
resp <- request("https://api.open-meteo.com/v1/forecast") %>%
  req_url_query(
    latitude = 48.21,
    longitude = 16.37,
    hourly = "temperature_2m,precipitation",
    timezone = "Europe/Vienna"
  ) %>%
  req_perform()

# Parse
weather_data <- resp_body_json(resp)

forecast <- tibble(
  time = weather_data$hourly$time %>% unlist() %>% ymd_hm(),
  temperature = weather_data$hourly$temperature_2m %>% unlist(),
  precipitation = weather_data$hourly$precipitation %>% unlist()
)

# Plot temperature
ggplot(forecast, aes(x = time, y = temperature)) +
  geom_line(color = "#00923f", linewidth = 0.6) +
  labs(
    title = "7-Day Temperature Forecast for Vienna",
    x = NULL,
    y = "Temperature (°C)"
  ) +
  theme_minimal()

# Bonus: plot precipitation
ggplot(forecast, aes(x = time, y = precipitation)) +
  geom_col(fill = "#201E50", width = 3000) +
  labs(
    title = "7-Day Precipitation Forecast for Vienna",
    x = NULL,
    y = "Precipitation (mm)"
  ) +
  theme_minimal()
```

The key difference from the Berlin example is the coordinates and timezone. The precipitation variable adds a second dimension to the analysis. Bar plots (`geom_col()`) are a natural choice for precipitation since it represents accumulated amounts per hour.

Robust Downloads

For scripts that run unattended or in production environments, it is worth building some resilience into the download process.

Retry Logic with httr2

Network requests can fail for transient reasons — a brief server overload, a momentary connectivity issue. The `req_retry()` function in {httr2} automatically retries failed requests:

```
resp <- request("https://api.open-meteo.com/v1/forecast") %>%
  req_url_query(latitude = 52.52, longitude = 13.41, current_weather = "true") %>%
  req_retry(max_tries = 3) %>%
  req_perform()
```

The `max_tries` argument sets how many attempts to make. By default, httr2 uses exponential backoff — it waits 1 second after the first failure, then 2 seconds, then 4 seconds, and so on. This avoids hammering a struggling server with rapid retries.

Checking File Integrity

For important downloads, verifying that the file arrived correctly is prudent. The `tools::md5sum()` function computes an MD5 hash that can be compared against a known checksum:

```
download.file(url, destfile = "data.csv")

# Compute MD5 hash of downloaded file
tools::md5sum("data.csv")

# Compare against expected hash
expected_hash <- "d41d8cd98f00b204e9800998ecf8427e"
actual_hash <- tools::md5sum("data.csv")

if (actual_hash != expected_hash) {
  warning("File hash mismatch — download may be corrupted!")
}
```

This is especially useful for large files where partial downloads can occur silently.

Simple Caching

A common pattern is to skip the download if the file already exists locally. This avoids unnecessary network traffic and speeds up repeated script runs:

```
dest <- "data/penguins.csv"

if (!file.exists(dest)) {
  download.file(url, destfile = dest)
  message("Downloaded: ", dest)
} else {
  message("Using cached file: ", dest)
}

data <- read_csv(dest)
```

For more sophisticated caching (e.g., re-downloading if the file is older than a week), one can check `file.info(dest)$mtime` against the current time.

curl::curl_download() as an Alternative

The {curl} package provides `curl_download()`, which offers a progress bar and more control over the connection:

```
# install.packages("curl")
curl::curl_download(url, destfile = "data.csv")
```

The {curl} package is a dependency of {httr2}, so it is typically already installed. It is a good choice when you want visible download progress for large files.

Common Pitfalls

Several issues arise regularly when downloading files in R, particularly on Windows and in corporate environments.

Binary Mode on Windows

As mentioned in the `download.file()` section, forgetting `mode = "wb"` when downloading binary files is perhaps the single most common mistake. A ZIP file downloaded in text mode will be corrupted — it may appear to have the right size but will fail to unzip. The fix is simple:

```
# Always use mode = "wb" for non-text files
download.file(url, destfile = "archive.zip", mode = "wb")
```

SSL/TLS Issues

Older R installations or systems with outdated certificates may fail on HTTPS URLs. The first thing to try is specifying the download method explicitly:

```
download.file(url, destfile = dest, method = "libcurl")
```

If that does not help, updating R and the system's CA certificate bundle usually resolves the issue.

Corporate Proxies

In corporate environments, internet access often goes through a proxy server. Both `download.file()` and {httr2} can be configured to use a proxy:

```
# For httr2
request(url) %>%
  req_proxy("http://proxy.company.com", port = 8080) %>%
  req_perform()

# For download.file, set environment variables before the call
Sys.setenv(
  http_proxy = "http://proxy.company.com:8080",
  https_proxy = "http://proxy.company.com:8080"
)
download.file(url, destfile = dest)
```

If you encounter download failures that work fine on a personal computer but fail at work, a proxy is the most likely cause. Contact your IT department for the correct proxy address and port.

Large File Downloads

For very large files (hundreds of megabytes or more), consider three adjustments: increase the timeout as shown earlier, use `curl::curl_download()` for its progress indicator, and verify the download with `tools::md5sum()`. Together, these prevent silent failures:

```
options(timeout = 600)  # 10 minutes

dest <- "large_dataset.zip"
curl::curl_download(url, destfile = dest)

# Verify
```

```
cat("MD5:", tools::md5sum(dest), "\n")
cat("Size:", file.size(dest) / 1e6, "MB\n")
```

Firewalls and Blocked URLs

Corporate networks may block access to certain domains entirely. This manifests as connection timeouts or “could not resolve host” errors. There is no programmatic fix for this — the only solution is to request that your IT department allowlist the required domains, or to download the files from an unrestricted network and transfer them manually.

Summary

This chapter covered the essential tools for programmatic file handling in R.

Key Takeaways

Task	Function / Package	Key Arguments
Simple download	`download.file()`	`url`, `destfile`, `mode = "wb"`
Temporary files	`tempfile()`, `tempdir()`	`fileext`, `pattern`
Auto-cleanup temps	`withr::local_tempfile()`	`fileext`
List ZIP contents	`unzip(..., list = TRUE)`	`zipfile`
Extract ZIP	`unzip()`	`zipfile`, `exdir`, `files`
Read from ZIP	`unz()`	`description`, `filename`
HTTP requests	`httr2::request()`	pipe with `req_*` and `resp_*`
Retry logic	`httr2::req_retry()`	`max_tries`, `backoff`
Download with progress	`curl::curl_download()`	`url`, `destfile`
File integrity	`tools::md5sum()`	`files`

The general principle is to start simple — `download.file()` with a `tempfile()` destination covers most use cases. Move to {httr2} when you need to interact with APIs, add headers, or handle authentication. Regardless of the approach, always use `mode = "wb"` for binary files on Windows and consider adding caching and integrity checks for production scripts.

Bibliography