13. Download und Entpacken

Programmatische Dateidownloads, Archive und HTTP-Anfragen

Dr. Paul Schmidt

Um alle in diesem Kapitel verwendeten Pakete zu installieren und zu laden, führt man folgenden Code aus:

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

library(tidyverse)
library(httr2)
```

Einleitung

In vielen Datenanalyse-Projekten besteht der erste Schritt darin, Daten aus einer externen Quelle zu beziehen — einem öffentlichen Repository, einem staatlichen Datenportal oder dem freigegebenen Ordner eines Kollegen. Der gängigste Ansatz ist, die Datei manuell über einen Webbrowser herunterzuladen und im Projektordner abzulegen. Das funktioniert zwar, erzeugt aber eine Lücke in der Reproduzierbarkeitskette: Wer die Analyse erneut ausführen möchte, muss wissen, woher die Datei stammt und wie man sie bekommt.

Wenn man Dateien programmatisch herunterlädt — direkt aus einem R-Skript heraus — schließt man diese Lücke. Die URL ist im Code selbst dokumentiert, wodurch die Datenquelle explizit und nachvollziehbar wird. Falls sich die Quelldaten ändern, lädt ein erneuter Skriptlauf automatisch die neueste Version herunter. Bei Workflows mit vielen Dateien oder zeitgesteuerten Abläufen (z.B. wöchentliche Berichte) wird manuelles Herunterladen schnell unpraktisch, während ein skriptbasierter Ansatz mühelos skaliert.

Über reine Downloads hinaus kann R auch mit Web-APIs interagieren, um Datenbanken abzufragen, Wetterdaten abzurufen oder auf jeden Dienst zuzugreifen, der eine programmatische Schnittstelle bereitstellt. Dieses Kapitel behandelt die wesentlichen Werkzeuge für all diese Aufgaben.

download.file()

Der einfachste Weg, eine Datei in R herunterzuladen, ist `download.file()`. Die Funktion nimmt eine URL und einen Zielpfad entgegen und speichert die Datei auf der Festplatte:

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

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

Nach dem Ausführen dieses Codes erscheint die Datei `penguins.csv` im Arbeitsverzeichnis. Man könnte sie dann wie gewohnt mit `read_csv("penguins.csv")` einlesen.

Binary Mode unter Windows

Ein kritisches Detail bei der Arbeit unter Windows ist das Argument `mode`. Standardmäßig verwendet `download.file()` den Textmodus, der Binärdateien wie ZIP-Archive, Excel-Tabellen oder Bilder beschädigen kann. Sicherheitshalber sollte man bei Nicht-Textdateien immer `mode = "wb"` (write binary) verwenden:

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

Für reine Textdateien wie CSV oder TSV funktioniert der Standardmodus problemlos. Im Zweifelsfall schadet `mode = "wb"` nie — es funktioniert auch bei Textdateien korrekt.

Timeout bei großen Dateien

Das Standard-Timeout für Downloads in R beträgt 60 Sekunden. Bei großen Dateien oder langsamen Verbindungen reicht das möglicherweise nicht aus. Man kann das Timeout über `options()` erhöhen:

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

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

Es empfiehlt sich, das Timeout am Anfang eines Skripts zu setzen, wenn große Downloads erwartet werden, anstatt das Problem erst mitten in der Ausführung zu entdecken.

HTTPS und Download-Methoden

Auf manchen Systemen kann `download.file()` bei HTTPS-URLs fehlschlagen. Die Angabe von `method = "libcurl"` löst dies in der Regel:

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

Bei modernen R-Installationen (4.2+) ist `"libcurl"` typischerweise bereits der Standard, sodass dies hauptsächlich für ältere Setups oder eingeschränkte Umgebungen relevant ist.

Übung: CSV herunterladen und einlesen

Der Palmer Penguins Datensatz ist als CSV-Datei unter folgender URL verfügbar:

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

1. Die Datei an einen temporären Speicherort herunterladen, wobei `tempfile(fileext = ".csv")` als Ziel verwendet wird.
2. Die heruntergeladene Datei mit `read_csv()` einlesen.
3. Das Ergebnis mit `glimpse()` inspizieren.

Lösung

```
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)
```

Die `glimpse()`-Ausgabe würde 344 Zeilen und 8 Spalten zeigen, darunter Art, Insel, Schnabelmaße, Flossenlänge, Körpermasse, Geschlecht und Jahr. Durch die Verwendung von `tempfile()` wird sichergestellt, dass der Download das Arbeitsverzeichnis nicht zumüllt — mehr dazu im nächsten Abschnitt.

Arbeiten mit temporären Dateien

Wenn heruntergeladene Daten nur für die Dauer eines Skripts benötigt werden, gibt es keinen Grund, sie dauerhaft zu speichern. R bietet eingebaute Funktionen zum Erstellen temporärer Dateien und Verzeichnisse, die automatisch aufgeräumt werden, wenn die R-Sitzung endet.

tempfile() und tempdir()

Die Funktion `tempfile()` erzeugt einen eindeutigen Dateipfad im temporären Verzeichnis von R. Die Datei existiert noch nicht — `tempfile()` erstellt lediglich einen Namen, der garantiert nicht mit bestehenden Dateien kollidiert:

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

```
[1] "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpkjH96Z\\file3014fe872b8"
```

```
tempfile()
```

```
[1] "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpkjH96Z\\file3014411455ce"
```

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

```
[1] "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpkjH96Z\\file30142ba573be.csv"
```

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

```
[1] "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpkjH96Z\
\penguins_30146d873d21.csv"
```

Das temporäre Verzeichnis selbst kann mit `tempdir()` inspiziert werden:

```
tempdir()
```

```
[1] "C:\\Users\\PAULSC~1\\AppData\\Local\\Temp\\RtmpkjH96Z"
```

Dieses Verzeichnis ist sitzungsspezifisch. Wenn R beendet wird, räumt das Betriebssystem es irgendwann auf. Temporäre Dateien sind daher ein guter Standard für Zwischen-Downloads — sie halten das Arbeitsverzeichnis sauber und sammeln sich nicht über die Zeit an.

withr::local_tempfile() für automatisches Aufräumen

Das Paket {withr} bietet `local_tempfile()`, das eine temporäre Datei erstellt. Die temporäre Datei wird automatisch gelöscht, wenn die umgebende Funktion zurückkehrt, auch wenn ein Fehler auftritt:

```
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
}
```

Für interaktive Nutzung ist `tempfile()` vollkommen ausreichend. Der `local_tempfile()`-Ansatz wird in Funktionen und Paketen wertvoll, wo man die Aufräumung auch bei Fehlern garantieren möchte.

Wann temporäre vs. permanente Dateien verwenden

Als Faustregel: Temporäre Dateien verwendet man, wenn der rohe Download nur ein Zwischenschritt ist (z.B. ein ZIP herunterladen, nur um dessen Inhalt zu extrahieren). Permanente Dateien verwendet man, wenn der Download selbst das Ergebnis ist, oder wenn man die Daten cachen möchte, um wiederholte Downloads zu vermeiden.

Archive entpacken

Viele Datenquellen verteilen Dateien als ZIP-Archive. R kann diese direkt ohne externe Werkzeuge verarbeiten.

ZIP-Datei inspizieren

Bevor man entpackt, ist es oft nützlich zu sehen, was ein ZIP-Archiv enthält. Die Funktion `unzip()` mit `list = TRUE` gibt einen Data Frame mit dem Archivinhalt zurück:

```
# 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 file3014622271d3.csv    263 2026-03-12 10:46:00
2  file30147a94698.csv    195 2026-03-12 10:46:00
```

Das Argument `list = TRUE` gibt einen Data Frame mit dem Archivinhalt zurück — Dateinamen, unkomprimierte Größen und Daten. Es ist gute Praxis, vor dem Entpacken zu inspizieren, besonders bei Archiven aus unbekannten Quellen.

Dateien extrahieren

Um alle Dateien zu extrahieren, ruft man `unzip()` mit dem Argument `exdir` auf, das angibt, wohin die extrahierten Dateien abgelegt werden sollen:

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

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

Um nur bestimmte Dateien zu extrahieren, verwendet man das Argument `files`:

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

Direkt aus einem ZIP lesen mit unz()

Für den häufigen Fall, dass ein ZIP eine einzelne CSV-Datei enthält, bietet R `unz()`, das eine Verbindung zu einer Datei innerhalb eines ZIPs erstellt, ohne auf die Festplatte zu extrahieren:

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

Das ist elegant und effizient — keine Zwischendateien, kein Aufräumen nötig. Es funktioniert gut mit `read_csv()`, `read_tsv()`, `read.csv()` und ähnlichen Funktionen, die Connections akzeptieren.

Übung: ZIP erstellen, inspizieren und entpacken

1. Zwei kleine CSV-Dateien aus eingebauten Datensätzen (`PlantGrowth` und `ToothGrowth`) erstellen und mit `zip()` in ein einziges ZIP-Archiv bündeln.
2. Den Archivinhalt mit `unzip(..., list = TRUE)` inspizieren.
3. Die Dateien in ein temporäres Verzeichnis extrahieren.
4. Eine der extrahierten CSV-Dateien mit `read_csv()` einlesen und die ersten Zeilen mit `head()` anzeigen.

Lösung

```
# 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  file301481eb10.csv    405 2026-03-12 10:46:00
2 file3014ebd5bda.csv    821 2026-03-12 10:46:00
```

```
# 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.
```

```
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
```

Die Verwendung von `tempfile()` als Extraktionsverzeichnis (anstelle von `tempdir()`) gibt einen sauberen, dedizierten Ordner ohne übrig gebliebene Dateien von anderen Operationen.

httr2 für HTTP-Anfragen

Während `download.file()` für einfache Downloads ausreicht, stellen viele moderne Datenquellen ihre Daten über Web-APIs (Application Programming Interfaces) bereit. APIs liefern Daten typischerweise im JSON-Format und können spezifische Header, Query-Parameter oder Authentifizierung erfordern. Das Paket {httr2} bietet eine saubere, pipe-freundliche Schnittstelle zum Erstellen und Ausführen von HTTP-Anfragen.

Das Request-Perform-Parse-Muster

Der Kern-Workflow in {httr2} folgt drei Schritten: die Anfrage aufbauen, ausführen und die Antwort parsen:

```
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
```

Die Funktion `request()` erstellt ein Request-Objekt aus einer Basis-URL. Die Funktion `req_url_query()` fügt Query-Parameter hinzu (der Teil nach `?` in einer URL). Schließlich sendet `req_perform()` die Anfrage und gibt ein Response-Objekt zurück.

Die Antwort inspizieren

Das Response-Objekt enthält den HTTP-Statuscode, Header und Body. Mehrere Hilfsfunktionen extrahieren diese Bestandteile:

```
# 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
```

Für das obige Open-Meteo-Beispiel gibt `resp_body_json()` eine verschachtelte Liste zurück. Die aktuelle Temperatur für Berlin wäre über `weather$current_weather$temperature` abrufbar.

Header hinzufügen

Manche APIs erfordern benutzerdefinierte Header, beispielsweise einen API-Key oder einen bestimmten Content-Type. Diese werden mit `req_headers()` hinzugefügt:

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

```
    Accept = "application/json"
  ) %>%
  req_perform()
```

> ! Wichtig
>
> API-Keys niemals direkt in Skripten hartcodieren, die in die Versionskontrolle eingecheckt werden. Stattdessen speichert man sie in Umgebungsvariablen und greift mit `Sys.getenv("MY_API_KEY")` darauf zu. So bleiben Zugangsdaten außerhalb der Git-Historie.

Vergleich mit download.file()

Für einfache Dateidownloads bleibt `download.file()` die unkomplizierteste Wahl. Man verwendet {httr2}, wenn man Folgendes braucht: Query-Parameter, benutzerdefinierte Header, Authentifizierung, JSON-Parsing, Fehlerbehandlung mit Retries oder die Interaktion mit REST-APIs. Die beiden Ansätze ergänzen sich, anstatt miteinander zu konkurrieren.

Praxisbeispiel - Wetterdaten von Open-Meteo

Um einen vollständigen API-Workflow zu veranschaulichen, fragen wir stündliche Temperaturdaten von der Open-Meteo API ab. Diese API ist kostenlos, erfordert keinen API-Key und liefert Wettervorhersagen sowie historische Daten für jeden Ort der Welt.

Die Anfrage aufbauen

Die Open-Meteo API erwartet Breitengrad, Längengrad und eine Angabe, welche Wettervariablen zurückgegeben werden sollen. Wir fordern stündliche Temperaturdaten für Berlin über die nächsten 7 Tage an:

```
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()
```

JSON in ein Tibble parsen

Die JSON-Antwort enthält verschachtelte Listen. Wir extrahieren die relevanten Teile und bauen daraus ein aufgeräumtes Tibble zusammen:

```
weather_data <- resp_body_json(resp)

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

forecast
```

Das Feld `time` kommt als Zeichenkette im ISO-8601-Format an, das `lubridate::ymd_hm()` (geladen mit {tidyverse}) in echte Datetime-Objekte umwandelt. Das Ergebnis ist ein sauberes Tibble mit einer Zeile pro Stunde.

Die Vorhersage visualisieren

Mit den Daten im Tidy-Format ist das Erstellen eines Plots unkompliziert:

```
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()
```

Dies würde ein Liniendiagramm erzeugen, das die stündliche Temperatur über die kommende Woche zeigt. Der Tagesgang (warm tagsüber, kühl nachts) ist typischerweise deutlich erkennbar.

Übung: Wetterdaten für eine andere Stadt abfragen

Die Open-Meteo API verwenden, um eine 7-Tage-Temperaturvorhersage (stündlich) für eine Stadt nach Wahl abzurufen.

1. Breitengrad und Längengrad der gewählten Stadt nachschlagen (z.B. über Google Maps oder latlong.net).
2. Die API-Anfrage aus dem obigen Beispiel mit den neuen Koordinaten anpassen.
3. Die Antwort in ein Tibble parsen und ein Liniendiagramm der Temperatur erstellen.
4. Bonus: `hourly = "temperature_2m,precipitation"` zur Abfrage hinzufügen und beide Variablen darstellen.

Lösung

Hier ein Beispiel für Wien (Breitengrad 48.21, Längengrad 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()
```

Der wesentliche Unterschied zum Berlin-Beispiel sind die Koordinaten und die Zeitzone. Die Niederschlagsvariable fügt der Analyse eine zweite Dimension hinzu. Balkendiagramme (`geom_col()`) sind eine natürliche Wahl für Niederschlag, da er akkumulierte Mengen pro Stunde darstellt.

Robuste Downloads

Für Skripte, die unbeaufsichtigt oder in Produktionsumgebungen laufen, lohnt es sich, etwas Widerstandsfähigkeit in den Download-Prozess einzubauen.

Retry-Logik mit httr2

Netzwerkanfragen können aus vorübergehenden Gründen fehlschlagen — eine kurzzeitige Serverüberlastung, ein momentaner Verbindungsausfall. Die Funktion `req_retry()` in {httr2} wiederholt fehlgeschlagene Anfragen automatisch:

```
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()
```

Das Argument `max_tries` legt fest, wie viele Versuche unternommen werden. Standardmäßig verwendet httr2 exponentielles Backoff — es wartet 1 Sekunde nach dem ersten Fehlschlag, dann 2 Sekunden, dann 4 Sekunden, und so weiter. Das vermeidet es, einen angeschlagenen Server mit schnellen Wiederholungsversuchen zu überlasten.

Dateiintegrität prüfen

Bei wichtigen Downloads ist es ratsam zu überprüfen, ob die Datei korrekt angekommen ist. Die Funktion `tools::md5sum()` berechnet einen MD5-Hash, der mit einer bekannten Prüfsumme verglichen werden kann:

```
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!")
}
```

Dies ist besonders nützlich bei großen Dateien, bei denen unvollständige Downloads unbemerkt auftreten können.

Einfaches Caching

Ein gängiges Muster ist es, den Download zu überspringen, wenn die Datei bereits lokal vorhanden ist. Das vermeidet unnötigen Netzwerkverkehr und beschleunigt wiederholte Skriptläufe:

```
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)
```

Für ausgefeilteres Caching (z.B. erneuter Download, wenn die Datei älter als eine Woche ist) kann man `file.info(dest)$mtime` mit der aktuellen Zeit vergleichen.

curl::curl_download() als Alternative

Das Paket {curl} bietet `curl_download()`, das einen Fortschrittsbalken und mehr Kontrolle über die Verbindung ermöglicht:

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

Das Paket {curl} ist eine Abhängigkeit von {httr2} und daher in der Regel bereits installiert. Es ist eine gute Wahl, wenn man bei großen Dateien einen sichtbaren Download-Fortschritt wünscht.

Häufige Fallstricke

Beim Herunterladen von Dateien in R treten regelmäßig einige Probleme auf, insbesondere unter Windows und in Unternehmensumgebungen.

Binary Mode unter Windows

Wie im Abschnitt zu `download.file()` erwähnt, ist das Vergessen von `mode = "wb"` beim Download von Binärdateien vermutlich der häufigste Fehler. Eine im Textmodus heruntergeladene ZIP-Datei ist beschädigt — sie mag die richtige Größe haben, lässt sich aber nicht entpacken. Die Lösung ist einfach:

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

SSL/TLS-Probleme

Ältere R-Installationen oder Systeme mit veralteten Zertifikaten können bei HTTPS-URLs fehlschlagen. Als Erstes sollte man die Download-Methode explizit angeben:

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

Falls das nicht hilft, löst in der Regel ein Update von R und des CA-Zertifikatsbundles des Systems das Problem.

Unternehmens-Proxies

In Unternehmensumgebungen läuft der Internetzugang oft über einen Proxy-Server. Sowohl `download.file()` als auch {httr2} lassen sich für die Nutzung eines Proxys konfigurieren:

```
# 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)
```

Wenn Downloads fehlschlagen, die auf einem privaten Rechner problemlos funktionieren, ist ein Proxy die wahrscheinlichste Ursache. Man sollte die IT-Abteilung nach der korrekten Proxy-Adresse und dem Port fragen.

Downloads großer Dateien

Bei sehr großen Dateien (mehrere hundert Megabyte oder mehr) empfehlen sich drei Anpassungen: das Timeout wie oben gezeigt erhöhen, `curl::curl_download()` wegen des Fortschrittsindikators verwenden und den Download mit `tools::md5sum()` verifizieren. Zusammen verhindern diese Maßnahmen stille Fehler:

```
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 und gesperrte URLs

Unternehmensnetzwerke können den Zugang zu bestimmten Domains komplett blockieren. Dies äußert sich als Verbindungs-Timeouts oder “could not resolve host”-Fehler. Es gibt dafür keine programmatische Lösung — man muss die IT-Abteilung bitten, die benötigten Domains freizuschalten, oder die Dateien über ein uneingeschränktes Netzwerk herunterladen und manuell übertragen.

Zusammenfassung

Dieses Kapitel hat die wesentlichen Werkzeuge für programmatische Dateiverarbeitung in R behandelt.

i Wichtige Erkenntnisse

Aufgabe	Funktion / Paket	Wichtige Argumente
Einfacher Download	`download.file()`	`url`, `destfile`, `mode = "wb"`
Temporäre Dateien	`tempfile()`, `tempdir()`	`fileext`, `pattern`
Auto-Cleanup Temps	`withr::local_tempfile()`	`fileext`
ZIP-Inhalt auflisten	`unzip(..., list = TRUE)`	`zipfile`
ZIP entpacken	`unzip()`	`zipfile`, `exdir`, `files`
Aus ZIP lesen	`unz()`	`description`, `filename`
HTTP-Anfragen	`httr2::request()`	Pipe mit `req_*` und `resp_*`
Retry-Logik	`httr2::req_retry()`	`max_tries`, `backoff`
Download mit Fortschritt	`curl::curl_download()`	`url`, `destfile`
Dateiintegrität	`tools::md5sum()`	`files`

Das allgemeine Prinzip lautet: einfach anfangen — `download.file()` mit einem `tempfile()`-Ziel deckt die meisten Anwendungsfälle ab. Man wechselt zu {httr2}, wenn man mit APIs interagieren, Header hinzufügen oder Authentifizierung handhaben muss. Unabhängig vom Ansatz sollte man unter Windows immer `mode = "wb"` für Binärdateien verwenden und für Produktionsskripte Caching und Integritätsprüfungen in Betracht ziehen.

Bibliography