Using ‘Copernicus Data Space Ecosystem’ API Wrapper

Contents

Introduction

The CDSE package for R was developed to allow access to the 'Copernicus Data Space Ecosystem' data and services from R. The 'Copernicus Data Space Ecosystem', deployed in 2023, offers access to the EO data collection from the Copernicus missions, with discovery and download capabilities and numerous data processing tools. In particular, the 'Sentinel Hub' API provides access to the multi-spectral and multi-temporal big data satellite imagery service, capable of fully automated, real-time processing and distribution of remote sensing data and related EO products. Users can use APIs to retrieve satellite data over their AOI and specific time range from full archives in a matter of seconds. When working on the application of EO where the area of interest is relatively small compared to the image tiles distributed by Copernicus (100 x 100 km), it allows to retrieve just the portion of the image of interest rather than downloading the huge tile image file and processing it locally. The goal of the CDSE package is to provide easy access to this functionality from R.

The main functions allow to search the catalog of available imagery from the Sentinel-1, Sentinel-2, Sentinel-3, and Sentinel-5 missions, and to process and download the images of an area of interest and a time range in various formats. Other functions might be added in subsequent releases of the package.

API authentication

Most of the API functions require OAuth2 authentication. The recommended procedure is to obtain an authentication client object from the `GetOAuthClient` function and pass it as the `client` argument to the functions requiring the authentication. For more detailed information, you are invited to consult the "`Before you start`" document.

```
id <- Sys.getenv("CDSE_ID")
secret <- Sys.getenv("CDSE_SECRET")
OAuthClient <- GetOAuthClient(id = id, secret = secret)
```

Note

In this document, the data.frames are output as tibbles since it renders better in PDF. However, all the functions produce standard data.frames.

Collections

We can get the list of all the imagery collections available in the `'Copernicus Data Space Ecosystem'`. By default, the list is formatted as a data.frame listing the main collection features. It is also possible to obtain the raw list with all information by setting the argument `as_data_frame` to `FALSE`.

```
collections <- GetCollections(as_data_frame = TRUE)
collections
#> # A tibble: 7 x 12
#>   id       title description since instrument   gsd bands constellation long.min
#>   <chr>    <chr> <chr>       <chr> <chr>      <int> <int> <chr>            <dbl>
#> 1 sentine~ Sent~ Sentinel 2~ 2015~ msi           10    13 sentinel-2        -180
#> 2 sentine~ Sent~ Sentinel 3~ 2016~ olci         300    NA <NA>              -180
#> 3 sentine~ Sent~ Sentinel 3~ 2016~ olci         300    21 <NA>              -180
#> 4 sentine~ Sent~ Sentinel 3~ 2016~ slstr       1000    11 <NA>              -180
#> 5 sentine~ Sent~ Sentinel 1~ 2014~ c-sar         NA    NA sentinel-1        -180
#> 6 sentine~ Sent~ Sentinel 2~ 2016~ msi           10    12 sentinel-2        -180
#> 7 sentine~ Sent~ Sentinel 5~ 2018~ tropomi     5500    NA <NA>              -180
#> # i 3 more variables: lat.min <dbl>, long.max <dbl>, lat.max <dbl>
```

Catalog search

The imagery catalog can be searched by spatial and temporal extent for every collection present in the `'Copernicus Data Space Ecosystem'`. For the spatial filter, you can provide either a `sf` or `sfc` object from the `sf` package, typically a (multi)polygon, describing the Area of Interest, or a numeric vector of four elements describing the bounding box of interest. For the temporal filter, you must specify the time range by either `Date` or `character` values that can be converted to date by `as.Date` function. Open intervals (one side only) can be obtained by providing the `NA` or `NULL` value for the corresponding argument.

```
dsn <- system.file("extdata", "luxembourg.geojson", package = "CDSE")
aoi <- sf::read_sf(dsn, as_tibble = FALSE)
images <- SearchCatalog(aoi = aoi, from = "2023-07-01", to = "2023-07-31",
    collection = "sentinel-2-l2a", with_geometry = TRUE, client = OAuthClient)
images
#> # A tibble: 70 x 12
#>    acquisitionDate tileCloudCover areaCoverage satellite  acquisitionTimestamp~1
#>    <date>                   <dbl>        <dbl> <chr>      <dttm>
#>  1 2023-07-31                98.9         1.84 sentinel-~ 2023-07-31 10:47:29
#>  2 2023-07-31                99.8        20.3  sentinel-~ 2023-07-31 10:47:25
#>  3 2023-07-31                99.7         5.93 sentinel-~ 2023-07-31 10:47:23
#>  4 2023-07-31                99.9        16.3  sentinel-~ 2023-07-31 10:47:14
#>  5 2023-07-31                99.9        92.5  sentinel-~ 2023-07-31 10:47:11
#>  6 2023-07-31                99.4        22.2  sentinel-~ 2023-07-31 10:47:09
#>  7 2023-07-28               100.          4.99 sentinel-~ 2023-07-28 10:37:28
#>  8 2023-07-28               100.          5.66 sentinel-~ 2023-07-28 10:37:27
#>  9 2023-07-28               100.          4.29 sentinel-~ 2023-07-28 10:37:21
#> 10 2023-07-28               100           6.86 sentinel-~ 2023-07-28 10:37:20
#> # i 60 more rows
#> # i abbreviated name: 1: acquisitionTimestampUTC
#> # i 7 more variables: acquisitionTimestampLocal <dttm>, sourceId <chr>,
#> #   long.min <dbl>, lat.min <dbl>, long.max <dbl>, lat.max <dbl>,
#> #   geometry <POLYGON [°]>
```

We can visualize the coverage of the area of interest by the satellite image tiles by plotting the footprints of the available images and showing the region of interest in red.

```
library(maps)
days <- range(as.Date(images$acquisitionDate))
maps::map(database = "world", col = "lightgrey", fill = TRUE, mar = c(0, 0, 4, 0),
    xlim = c(3, 9), ylim = c(47.5, 51.5))
plot(sf::st_geometry(aoi), add = TRUE, col = "red", border = FALSE)
plot(sf::st_geometry(images), add = TRUE)
title(main = sprintf("AOI coverage by image tiles for period %s",
    paste(days, collapse = " / ")), line = 1L, cex.main = 0.75)
```

Some tiles cover only a small fraction of the area of interest, while others cover almost the entire area.

```
summary(images$areaCoverage)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   1.845   5.603  15.113  19.758  20.346  92.463
```

The tile number can be obtained from the image attribute `sourceId`, as explained here. We can therefore summarize the distribution of area coverage by tile number, and see which tiles provide the best coverage of the AOI.

```
tileNumber <- substring(images$sourceId, 39, 44)
by(images$areaCoverage, INDICES = tileNumber, FUN = summary)
```

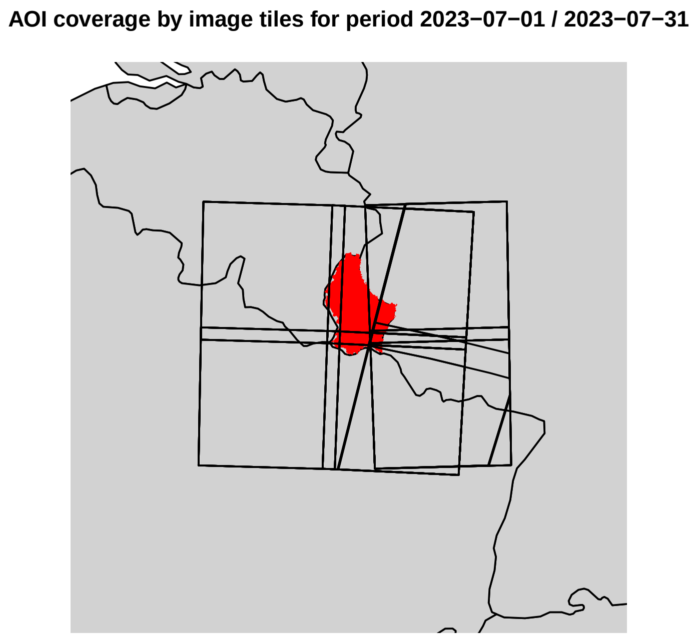

Figure 1: Luxembourg image tiles coverage

```
#> tileNumber: T31UFQ
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   1.845   1.845   1.845   1.845   1.845   1.845
#> ------------------------------------------------------------
#> tileNumber: T31UFR
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   16.32   16.32   16.32   16.32   16.32   16.32
#> ------------------------------------------------------------
#> tileNumber: T31UGQ
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   4.294   4.909  12.706  12.586  20.346  20.346
#> ------------------------------------------------------------
#> tileNumber: T31UGR
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   6.855  15.607  54.299  53.426  92.463  92.463
#> ------------------------------------------------------------
#> tileNumber: T32ULA
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   6.169  14.950  18.814  17.972  22.236  22.236
#> ------------------------------------------------------------
#> tileNumber: T32ULV
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   4.944   5.603   5.819   5.723   5.934   5.934
```

Catalog by season

Sometimes one can be interested in only a given period of each year, for example, the images taken during the summer months (June to August). We can filter an existing image catalog *a posteriori* using the `SeasonalFilter` function.

```
dsn <- system.file("extdata", "centralpark.geojson", package = "CDSE")
aoi <- sf::read_sf(dsn, as_tibble = FALSE)
images <- SearchCatalog(aoi = aoi, from = "2021-01-01", to = "2023-12-31",
    collection = "sentinel-2-l2a", with_geometry = FALSE, filter = "eo:cloud_cover < 5",
    client = OAuthClient)
dim(images)
#> [1] 77 10
summer_images <- SeasonalFilter(images, from = "2021-06-01", to = "2023-08-31")
dim(summer_images)
#> [1] 14 10
```

It is also possible to query the API directly on the desired seasonal periods by using a vectorized version of the `SearchCatalog` function. The vectorized versions allow running a series of queries having the same parameter values except for either time range, AOI, or the bounding box parameters, using `lapply` or similar function, and thus potentially also using the parallel processing.

```
dsn <- system.file("extdata", "centralpark.geojson", package = "CDSE")
aoi <- sf::read_sf(dsn, as_tibble = FALSE)
seasons <- SeasonalTimerange(from = "2021-06-01", to = "2023-08-31")
lst_summer_images <- lapply(seasons, SearchCatalogByTimerange, aoi = aoi,
    collection = "sentinel-2-l2a", filter = "eo:cloud_cover < 5", with_geometry = FALSE,
    client = OAuthClient)
summer_images <- do.call(rbind, lst_summer_images)
dim(summer_images)
#> [1] 14 10
summer_images <- summer_images[rev(order(summer_images$acquisitionDate)), ]
row.names(summer_images) <- NULL
summer_images
#> # A tibble: 14 x 10
#>    acquisitionDate tileCloudCover satellite   acquisitionTimestampUTC
#>    <date>                   <dbl> <chr>       <dttm>
#>  1 2023-08-20                0.03 sentinel-2a 2023-08-20 15:51:57
#>  2 2023-07-31                3.96 sentinel-2a 2023-07-31 15:51:56
#>  3 2023-07-26                0.89 sentinel-2b 2023-07-26 15:51:56
#>  4 2023-07-11                4.49 sentinel-2a 2023-07-11 15:51:56
#>  5 2023-06-01                0    sentinel-2a 2023-06-01 15:51:54
#>  6 2022-08-25                1.69 sentinel-2a 2022-08-25 15:52:03
#>  7 2022-08-03                4.58 sentinel-2b 2022-08-03 16:01:51
#>  8 2022-07-19                1.37 sentinel-2a 2022-07-19 16:01:58
#>  9 2022-07-11                4.89 sentinel-2b 2022-07-11 15:51:57
#> 10 2022-06-19                1.91 sentinel-2a 2022-06-19 16:01:58
#> 11 2022-06-06                1    sentinel-2a 2022-06-06 15:51:58
#> 12 2022-06-04                2.76 sentinel-2b 2022-06-04 16:01:47
#> 13 2021-06-16                4.74 sentinel-2b 2021-06-16 15:51:51
#> 14 2021-06-06                0.38 sentinel-2b 2021-06-06 15:51:52
#> # i 6 more variables: acquisitionTimestampLocal <dttm>, sourceId <chr>,
#> #   long.min <dbl>, lat.min <dbl>, long.max <dbl>, lat.max <dbl>
```

Scripts

As we shall see in the examples below, we have to provide a `script` argument to the `GetArchiveImage` function.

An evalscript (or "custom script") is a piece of JavaScript code that defines how the satellite data shall be processed by the API and what values the service shall return. It is a required part of any request involving data processing, such as retrieving an image of the area of interest.

The evaluation scripts can use any JavaScript function or language structures, along with certain utility functions provided by the API for user convenience. Chrome V8 JavaScript engine is used for running the evalscripts.

The evaluation scripts are passed as the `script` argument to the `GetArchiveImage` function. It has to be either a character string containing the evaluation script or the name of the file containing the script. The `scripts` folder of this package contains a few examples of evaluation scripts.

It is beyond the scope of this document to provide guidance for writing scripts, we encourage users to consult the API Beginners Guide and Evalscript (custom script) documentation. You can find a big collection of custom scripts that you can readily use in this repository.

Retrieving images

Retrieving AOI satellite image as a raster object

One of the most important features of the API is its ability to extract only the part of the images covering the area of interest. If the AOI is small as in the example below, this is a significant gain in efficiency (download, local processing) compared to getting the whole tile image and processing it locally.

```
dsn <- system.file("extdata", "centralpark.geojson", package = "CDSE")
aoi <- sf::read_sf(dsn, as_tibble = FALSE)
images <- SearchCatalog(aoi = aoi, from = "2021-05-01", to = "2021-05-31",
    collection = "sentinel-2-l2a", with_geometry = TRUE, client = OAuthClient)
images
#> # A tibble: 12 x 12
#>    acquisitionDate tileCloudCover areaCoverage satellite  acquisitionTimestamp~1
#>    <date>                   <dbl>        <dbl> <chr>      <dttm>
#>  1 2021-05-30              100            100 sentinel-~ 2021-05-30 16:01:47
#>  2 2021-05-27               16.3          100 sentinel-~ 2021-05-27 15:51:51
#>  3 2021-05-25               26.5          100 sentinel-~ 2021-05-25 16:01:47
#>  4 2021-05-22              100            100 sentinel-~ 2021-05-22 15:51:51
#>  5 2021-05-20               24.3          100 sentinel-~ 2021-05-20 16:01:47
#>  6 2021-05-17                7.17         100 sentinel-~ 2021-05-17 15:51:50
#>  7 2021-05-15               28.2          100 sentinel-~ 2021-05-15 16:01:47
#>  8 2021-05-12                1.35         100 sentinel-~ 2021-05-12 15:51:50
#>  9 2021-05-10               92.7          100 sentinel-~ 2021-05-10 16:01:45
#> 10 2021-05-07               89.6          100 sentinel-~ 2021-05-07 15:51:48
#> 11 2021-05-05              100.           100 sentinel-~ 2021-05-05 16:01:45
#> 12 2021-05-02               78            100 sentinel-~ 2021-05-02 15:51:48
#> # i abbreviated name: 1: acquisitionTimestampUTC
#> # i 7 more variables: acquisitionTimestampLocal <dttm>, sourceId <chr>,
#> #   long.min <dbl>, lat.min <dbl>, long.max <dbl>, lat.max <dbl>,
#> #   geometry <POLYGON [°]>
summary(images$areaCoverage)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>     100     100     100     100     100     100
```

As the area is small, it is systematically fully covered by all available images. We shall select the date with the least cloud cover, and retrieve the NDVI values as a `SpatRaster` from package `terra`. This allows further processing of the data, as shown below by replacing all negative values with zero. The size of the pixels is specified directly by the `resolution` argument. We are also adding a 100-meter `buffer` around the area of interest and `mask`ing the pixels outside of the AOI.

```
day <- images[order(images$tileCloudCover), ]$acquisitionDate[1]
script_file <- system.file("scripts", "NDVI_float32.js", package = "CDSE")
ras <- GetImage(aoi = aoi, time_range = day, script = script_file,
    collection = "sentinel-2-l2a", format = "image/tiff", mosaicking_order = "leastCC",
    resolution = 10, mask = TRUE, buffer = 100, client = OAuthClient)
ras
#> class       : SpatRaster
#> dimensions  : 383, 355, 1  (nrow, ncol, nlyr)
#> resolution  : 0.0001003292, 0.0001003292  (x, y)
#> extent      : -73.98355, -73.94794, 40.76322, 40.80165  (xmin, xmax, ymin, ymax)
#> coord. ref. : lon/lat WGS 84 (EPSG:4326)
#> source(s)   : memory
#> name        : file930d6f5abb9c
#> min value   :       -0.5069648
```

```
#> max value   :         0.9507549
ras[ras < 0] <- 0
terra::plot(ras, main = paste("Central Park NDVI on", day), cex.main = 0.75,
    col = colorRampPalette(c("darkred", "yellow", "darkgreen"))(99))
```

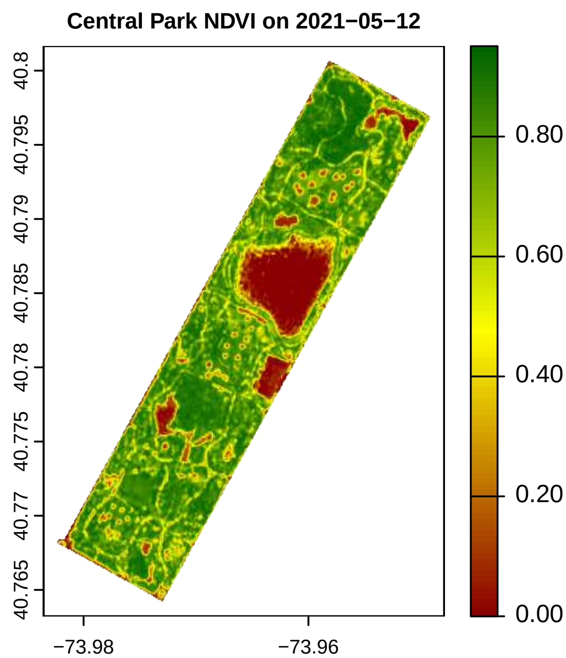

Figure 2: Central Park NDVI raster

Retrieving AOI satellite image as an image file

If we don't want to process the satellite image locally but simply use it as an image file (to include in a report or a Web page, for example), we can use the appropriate script that will render a three-band raster for RGB layers (or one for black-and-white image). Here we specify the area of interest by its bounding box instead of the exact geometry. We also demonstrate that the evaluation script can be passed as a single character string, and provide the number of pixels in the output image rather than the size of individual pixels - it makes more sense if the image is intended for display and not processing.

```
bbox <- as.numeric(sf::st_bbox(aoi))
script_text <- paste(readLines(system.file("scripts", "TrueColorS2L2A.js",
    package = "CDSE")), collapse = "\n")
cat(c(readLines(system.file("scripts", "TrueColorS2L2A.js", package = "CDSE"), n = 15),
    "..."), sep = "\n")
#> //VERSION=3
#> //Optimized Sentinel-2 L2A True Color
#>
#> function setup() {
#>   return {
#>     input: ["B04", "B03", "B02", "dataMask"],
#>     output: { bands: 4 }
#>   };
#> }
```

```
#>
#> function evaluatePixel(smp) {
#>   const rgbLin = satEnh(sAdj(smp.B04), sAdj(smp.B03), sAdj(smp.B02));
#>   return [sRGB(rgbLin[0]), sRGB(rgbLin[1]), sRGB(rgbLin[2]), smp.dataMask];
#> }
#>
#> ...
png <- tempfile("img", fileext = ".png")
GetImage(bbox = bbox, time_range = day, script = script_text,
    collection = "sentinel-2-l2a", file = png, format = "image/png",
    mosaicking_order = "leastCC", pixels = c(600, 950), client = OAuthClient)
terra::plotRGB(terra::rast(png))
```

Figure 3: Central Park image as PNG file

Retrieving a series of images in a batch

It often happens that one is interested in acquiring a series of images of a particular zone (AOI or bounding box) for several dates, or the images of different areas of interest for the same date (probably located close to each other so that they are visited on the same day). The `GetImageBy*` functions (`GetImageByTimerange`, `GetImageByAOI`, `GetImageByBbox`) facilitate this task as they are specifically crafted for being called from a `lapply`-like function, and thus potentially be executed in parallel. We shall illustrate how to do this with an example.

```
dsn <- system.file("extdata", "centralpark.geojson", package = "CDSE")
aoi <- sf::read_sf(dsn, as_tibble = FALSE)
images <- SearchCatalog(aoi = aoi, from = "2023-01-01", to = "2023-12-31",
                        collection = "sentinel-2-l2a", with_geometry = TRUE,
                        filter = "eo:cloud_cover < 5", client = OAuthClient)
# Get the day with the minimal cloud cover for every month -----------------------------
tmp1 <- images[, c("tileCloudCover", "acquisitionDate")]
tmp1$month <- lubridate::month(images$acquisitionDate)
agg1 <- stats::aggregate(tileCloudCover ~ month, data = tmp1, FUN = min)
tmp2 <- merge.data.frame(agg1, tmp1, by = c("month", "tileCloudCover"), sort = FALSE)
```

```r
# in case of ties, get an arbitrary date (here the smallest acquisitionDate,
# could also be the biggest)
agg2 <- stats::aggregate(acquisitionDate ~ month, data = tmp2, FUN = min)
monthly <- merge.data.frame(agg2, tmp2, by = c("acquisitionDate", "month"), sort = FALSE)
days <- monthly$acquisitionDate
# Retrieve images in parallel ---------------------------------------------------------
script_file <- system.file("scripts", "NDVI_float32.js", package = "CDSE")
tmp_folder <- tempfile("dir")
if (!dir.exists(tmp_folder)) dir.create(tmp_folder)
cl <- parallel::makeCluster(4)
ans <- parallel::clusterExport(cl, list("tmp_folder"), envir = environment())
ans <- parallel::clusterEvalQ(cl, {library(CDSE)})
lstRast <- parallel::parLapply(cl, days, fun = function(x, ...) {
    GetImageByTimerange(x, file = sprintf("%s/img_%s.tiff", tmp_folder, x), ...)},
    aoi = aoi, collection = "sentinel-2-l2a", script = script_file,
    format = "image/tiff", mosaicking_order = "mostRecent", resolution = 10,
    buffer = 0, mask = TRUE, client = OAuthClient)
parallel::stopCluster(cl)
# Plot the images ---------------------------------------------------------------------
par(mfrow = c(3, 4))
ans <- sapply(seq_along(days), FUN = function(i) {
    ras <- terra::rast(lstRast[[i]])
    day <- days[i]
    ras[ras < 0] <- 0
    terra::plot(ras, main = paste("Central Park NDVI on", day), range = c(0, 1),
        cex.main = 0.7, pax = list(cex.axis = 0.5), plg = list(cex = 0.5),
        col = colorRampPalette(c("darkred", "yellow", "darkgreen"))(99))
    })
```

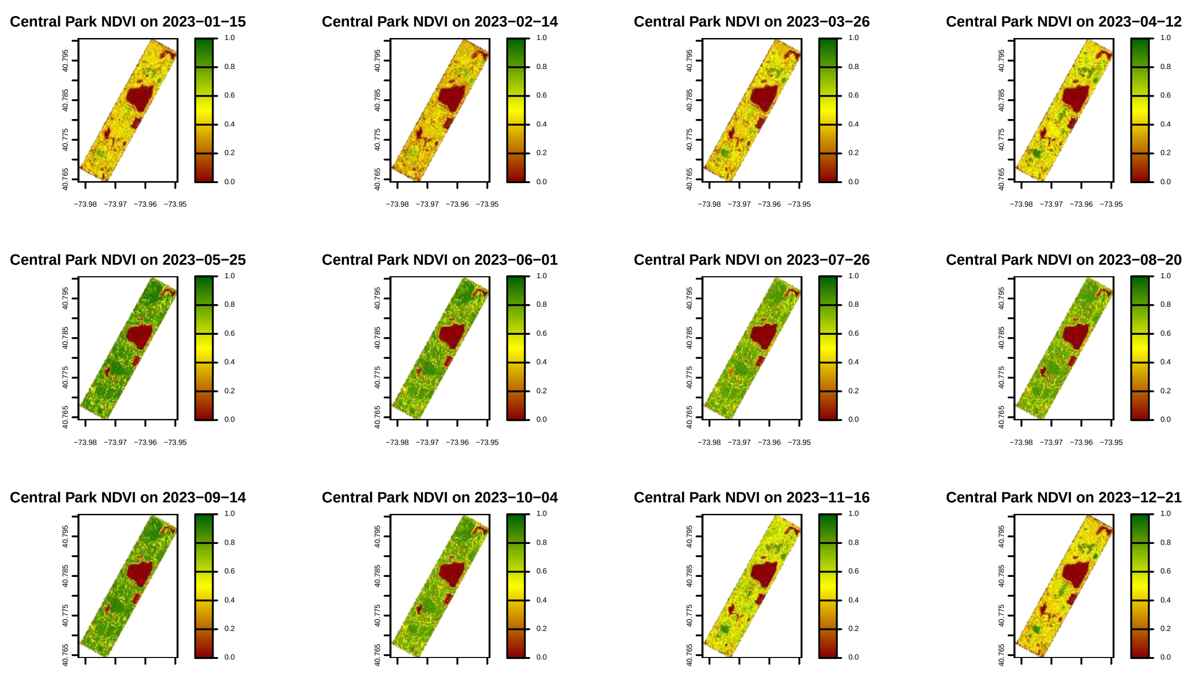

Figure 4: Central Park monthly NDVI

In this particular example, parallelization is not necessarily beneficial as we are retrieving only 12 images, but for a large number of images, it can significantly reduce the execution time. Also note that we have used the `filter` argument of the `SearchCatalog` function to limit the list of images to the tiles having cloud cover $<$ 5%.

Retrieving statistics

If you are only interested in calculating the average value (or some other statistic) of some index or just the raw band values, the Statistical API enables you to get statistics calculated based on satellite imagery without having to download images. You need to specify your area of interest, time period, evalscript, and which statistical measures should be calculated. The requested statistics are returned as a `data.frame` or as a `list`.

Statistical evalscripts

All general rules for building evalscripts apply. However, there are some specifics when using evalscripts with the Statistical API:

- The `evaluatePixel()` function must, in addition to other output, always return also `dataMask` output. This output defines which pixels are excluded from calculations. For more details and an example, see here.
- The default value of `sampleType` is `FLOAT32`.
- The output.bands parameter in the `setup()` function can be an array. This makes it possible to specify custom names for the output bands and different output `dataMask` for different outputs.

Retrieving simple statistics

Besides the time range, you have to specify the way you want the values to be aggregated over time. For this you use the `aggregation_period` and `aggregation_unit` arguments. The `aggregation_unit` must be one of `day`, `week`, `month` or `year`, and the `aggregation_period` providing the number of `aggregation_units` (days, weeks, ...) over which the statistics are calculated. The default values are "1" and "day", producing the daily statistics. If the last interval in the given time range isn't divisible by the provided aggregation interval, you can skip the last interval (default behavior), shorten the last interval so that it ends at the end of the provided time range, or extend the last interval over the end of the provided time range so that all intervals are of equal duration. This is controlled by the value of the `lastIntervalBehavior` argument.

```r
dsn <- system.file("extdata", "centralpark.geojson", package = "CDSE")
aoi <- sf::read_sf(dsn, as_tibble = FALSE)
script_file <- system.file("scripts", "NDVI_CLOUDS_STAT.js", package = "CDSE")
daily_stats <- GetStatistics(aoi = aoi, time_range = c("2023-07-01", "2023-07-31"),
    collection = "sentinel-2-l2a", script = script_file, mosaicking_order = "leastCC",
    resolution = 100, aggregation_period = 1, client = OAuthClient)
weekly_stats <- GetStatistics(aoi = aoi, time_range = c("2023-07-01", "2023-07-31"),
    collection = "sentinel-2-l2a", script = script_file,mosaicking_order = "leastCC",
    resolution = 100, aggregation_period = 7, client = OAuthClient)
weekly_stats_extended <- GetStatistics(aoi = aoi,
    time_range = c("2023-07-01", "2023-07-31"), collection = "sentinel-2-l2a",
    script = script_file, mosaicking_order = "leastCC", resolution = 100,
    aggregation_period = 1, aggregation_unit = "w", lastIntervalBehavior = "EXTEND",
    client = OAuthClient)
daily_stats
#> # A tibble: 26 x 9
#>    date       output band    min    mean     max   stDev sampleCount noDataCount
#>    <date>     <chr>  <chr> <dbl>   <dbl>   <dbl>   <dbl>       <int>       <int>
#>  1 2023-07-01 stati~ ndvi~     0 2.10e-1 6.51e-1 1.70e-1        1120         704
#>  2 2023-07-01 stati~ clou~     0 2.10e+1 5.9 e+1 1.41e+1        1120         704
#>  3 2023-07-04 stati~ ndvi~     0 1.15e-3 7.14e-2 6.54e-3        1120         704
#>  4 2023-07-04 stati~ clou~    65 6.5 e+1 6.5 e+1 0              1120         704
#>  5 2023-07-06 stati~ ndvi~     0 5.91e-1 9.37e-1 3.13e-1        1120         704
```

```
#>  6 2023-07-06 stati~ clou~     0 4.09e-2 1    e+0 1.98e-1        1120          704
#>  7 2023-07-09 stati~ ndvi~     0 2.07e-4 2.91e-3 5.08e-4        1120          704
#>  8 2023-07-09 stati~ clou~    65 6.5 e+1 6.5 e+1 0              1120          704
#>  9 2023-07-11 stati~ ndvi~     0 5.82e-1 9.16e-1 3.05e-1        1120          704
#> 10 2023-07-11 stati~ clou~     0 3.13e-2 1    e+0 1.74e-1        1120          704
#> # i 16 more rows
weekly_stats
#> # A tibble: 8 x 10
#>   from       to         output    band    min   mean    max  stDev sampleCount
#>   <date>     <date>     <chr>     <chr> <dbl>  <dbl>  <dbl>  <dbl>       <int>
#> 1 2023-07-01 2023-07-07 statisti~ ndvi~     0  0.591    0.937  0.313        1120
#> 2 2023-07-01 2023-07-07 statisti~ clou~     0  0.0409   1      0.198        1120
#> 3 2023-07-08 2023-07-14 statisti~ ndvi~     0  0.582    0.916  0.305        1120
#> 4 2023-07-08 2023-07-14 statisti~ clou~     0  0.0313   1      0.174        1120
#> 5 2023-07-15 2023-07-21 statisti~ ndvi~     0  0.342    0.872  0.237        1120
#> 6 2023-07-15 2023-07-21 statisti~ clou~     0 19.0    100     27.5          1120
#> 7 2023-07-22 2023-07-28 statisti~ ndvi~     0  0.550    0.860  0.282        1120
#> 8 2023-07-22 2023-07-28 statisti~ clou~     0  0.0409   1      0.198        1120
#> # i 1 more variable: noDataCount <int>
weekly_stats_extended
#> # A tibble: 10 x 10
#>    from       to         output   band    min   mean    max  stDev sampleCount
#>    <date>     <date>     <chr>    <chr> <dbl>  <dbl>  <dbl>  <dbl>       <int>
#>  1 2023-07-01 2023-07-07 statist~ ndvi~     0  0.591    0.937  0.313        1120
#>  2 2023-07-01 2023-07-07 statist~ clou~     0  0.0409   1      0.198        1120
#>  3 2023-07-08 2023-07-14 statist~ ndvi~     0  0.582    0.916  0.305        1120
#>  4 2023-07-08 2023-07-14 statist~ clou~     0  0.0313   1      0.174        1120
#>  5 2023-07-15 2023-07-21 statist~ ndvi~     0  0.342    0.872  0.237        1120
#>  6 2023-07-15 2023-07-21 statist~ clou~     0 19.0    100     27.5          1120
#>  7 2023-07-22 2023-07-28 statist~ ndvi~     0  0.550    0.860  0.282        1120
#>  8 2023-07-22 2023-07-28 statist~ clou~     0  0.0409   1      0.198        1120
#>  9 2023-07-29 2023-08-04 statist~ ndvi~     0  0.575    0.904  0.300        1120
#> 10 2023-07-29 2023-08-04 statist~ clou~     0  0.0240   1      0.153        1120
#> # i 1 more variable: noDataCount <int>
```

In this example we have demonstrated that a week can be specified as either 7 days or 1 week.

Retrieving statistics with percentiles

Besides the basic statistics (min, max, mean, stDev), one can also request to compute the percentiles. If the percentiles requested are 25, 50, and 75, the corresponding output is renamed 'q1', 'median', and 'q3'.

```
daily_stats <- GetStatistics(aoi = aoi, time_range = c("2023-07-01", "2023-07-31"),
    collection = "sentinel-2-l2a", script = script_file, mosaicking_order = "leastCC",
    resolution = 100, aggregation_period = 1, percentiles = c(25, 50, 75),
    client = OAuthClient)
head(daily_stats, n = 10)
#> # A tibble: 10 x 12
#>    date       output   band    min      q1 median    mean     q3     max   stDev
#>    <date>     <chr>    <chr> <dbl>   <dbl>  <dbl>   <dbl>  <dbl>   <dbl>   <dbl>
#>  1 2023-07-01 statist~ ndvi~     0  0.0619   0.185 2.10e-1  0.344 6.51e-1 1.70e-1
#>  2 2023-07-01 statist~ clou~     0  9       20     2.10e+1 32     5.9 e+1 1.41e+1
#>  3 2023-07-04 statist~ ndvi~     0  0        0     1.15e-3  0     7.14e-2 6.54e-3
#>  4 2023-07-04 statist~ clou~    65 65       65     6.5 e+1 65     6.5 e+1 0
```

```
#>  5 2023-07-06 statist~ ndvi~     0  0.393   0.729 5.91e-1  0.829 9.37e-1 3.13e-1
#>  6 2023-07-06 statist~ clou~     0  0       0     4.09e-2  0     1   e+0 1.98e-1
#>  7 2023-07-09 statist~ ndvi~     0  0       0     2.07e-4  0     2.91e-3 5.08e-4
#>  8 2023-07-09 statist~ clou~    65 65      65     6.5 e+1 65     6.5 e+1 0
#>  9 2023-07-11 statist~ ndvi~     0  0.383   0.719 5.82e-1  0.816 9.16e-1 3.05e-1
#> 10 2023-07-11 statist~ clou~     0  0       0     3.13e-2  0     1   e+0 1.74e-1
#> # i 2 more variables: sampleCount <int>, noDataCount <int>
weekly_stats <- GetStatistics(aoi = aoi, time_range = c("2023-07-01", "2023-07-31"),
    collection = "sentinel-2-l2a", script = script_file,mosaicking_order = "leastCC",
    resolution = 100, aggregation_period = 7, percentiles = seq(10, 90, by = 10),
    client = OAuthClient)
head(weekly_stats, n = 10)
#> # A tibble: 8 x 19
#>   from       to         output    band    min P.10.0 P.20.0 P.30.0 P.40.0 P.50.0
#>   <date>     <date>     <chr>     <chr> <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
#> 1 2023-07-01 2023-07-07 statisti~ ndvi~     0 0       0.239  0.527  0.649  0.729
#> 2 2023-07-01 2023-07-07 statisti~ clou~     0 0       0      0      0      0
#> 3 2023-07-08 2023-07-14 statisti~ ndvi~     0 0       0.247  0.528  0.643  0.719
#> 4 2023-07-08 2023-07-14 statisti~ clou~     0 0       0      0      0      0
#> 5 2023-07-15 2023-07-21 statisti~ ndvi~     0 0.0311  0.110  0.184  0.245  0.315
#> 6 2023-07-15 2023-07-21 statisti~ clou~     0 0       0      0      2      4
#> 7 2023-07-22 2023-07-28 statisti~ ndvi~     0 0       0.235  0.517  0.613  0.680
#> 8 2023-07-22 2023-07-28 statisti~ clou~     0 0       0      0      0      0
#> # i 9 more variables: mean <dbl>, P.60.0 <dbl>, P.70.0 <dbl>, P.80.0 <dbl>,
#> #   P.90.0 <dbl>, max <dbl>, stDev <dbl>, sampleCount <int>, noDataCount <int>
```

Retrieving a series of statistics in a batch

Just as when retrieving satellite images, one can be interested in acquiring a series of statistics for a particular zone (AOI or bounding box) for several dates, or the statistics of different zones for the same periods. The `GetStatisticsBy*` functions (`GetStatisticsByTimerange`, `GetStatisticsByAOI`, `GetStatisticsByBbox`) facilitate this task as they are specifically crafted for being called from a `lapply`-like function, and thus potentially be executed in parallel. The following example illustrates how to do this.

```
dsn <- system.file("extdata", "centralpark.geojson", package = "CDSE")
aoi <- sf::read_sf(dsn, as_tibble = FALSE)
script_file <- system.file("scripts", "NDVI_dataMask_float32.js", package = "CDSE")
seasons <- SeasonalTimerange(from = "2020-06-01", to = "2023-08-31")
lst_stats <- lapply(seasons, GetStatisticsByTimerange, aoi = aoi,
    collection = "sentinel-2-l2a", script = script_file, mosaicking_order = "leastCC",
    resolution = 100, aggregation_period = 7L, client = OAuthClient)
weekly_stats <- do.call(rbind, lst_stats)
weekly_stats <- weekly_stats[order(weekly_stats$from), ]
row.names(weekly_stats) <- NULL
head(weekly_stats, n = 5)
#> # A tibble: 5 x 10
#>   from       to         output     band         min  mean   max stDev sampleCount
#>   <date>     <date>     <chr>      <chr>      <dbl> <dbl> <dbl> <dbl>       <int>
#> 1 2020-06-01 2020-06-07 statistics ndvi_val~     0 0.510 0.868 0.291        1120
#> 2 2020-06-08 2020-06-14 statistics ndvi_val~     0 0.573 0.943 0.305        1120
#> 3 2020-06-15 2020-06-21 statistics ndvi_val~     0 0.621 1     0.321        1120
#> 4 2020-06-22 2020-06-28 statistics ndvi_val~     0 0.459 0.850 0.274        1120
#> 5 2020-06-29 2020-07-05 statistics ndvi_val~     0 0.599 0.930 0.301        1120
#> # i 1 more variable: noDataCount <int>
```

Copernicus Data Space Ecosystem services status

If you encounter any connection issues while using this package, please check your internet connection first. If your internet connection is working fine, you can also check the status of the Copernicus Data Space Ecosystem services by visiting this webpage. It provides a quasi real-time status of the various services provided. Once you are on the webpage, scroll down to Sentinel Hub, and pay particular attention to the Process API (used for retrieving images), Catalog API (used for catalog searches), and Statistical API (used for retrieving statistics).