BaseSpaceR

Adrian Alexa

October 24, 2025
aalexa@illumina.com

Contents

1 Introduction **2**

1.1 Developing BaseSpace applications 2

2 Installation **3**

3 BaseSpace data model **3**

3.1 `Item` objects 4

3.2 `Collection` objects 4

3.3 `Response` objects 4

4 Sample Session - Quick API guide **4**

4.1 Users 5

4.2 Genomes 7

4.3 Runs 11

4.4 Projects 14

4.5 Files 16

5 Authentication **18**

5.1 The OAuth v2 workflow 19

5.2 Using a pre-generated access token 20

6 Session Information **22**

1 Introduction

The `BaseSpaceR` package provides a complete R interface to Illumina's BaseSpace REST API, enabling the fast development of data analysis and visualisation tools. Besides providing an easy to use set of tools for manipulating the data from BaseSpace, it also facilitates the access to R's rich environment of statistical and data analysis open source tools.

Features include:

- Persistent connection with the REST server.
- Support for the REST API query parameters.
- Vectorized operations in line with the R semantic. Allows for queries across multiple Projects, Samples, AppResults, Files, etc.
- S4 class system used to represent the BaseSpace data model.
- Templates for uploading and creating AppResults *[under development]*.
- Integration with Bioconductor libraries and data containers *[under development]*.
- Portability on most platforms: Linux, Windows and Mac OS X.

BaseSpace is a proprietary cloud platform developed by Illumina for storing, analyzing and sharing genetic data. General information about BaseSpace can be found at basespace.illumina.com and resources for BaseSpace developers can be found at developer.basespace.illumina.com.

1.1 Developing BaseSpace applications

The following briefly presents the steps necessary to get started with `BaseSpaceR` and the process of creating a new App. At the same time it provides an outlook of the sections to follow. It is essential for the developer to read through BaseSpace's Getting Started guide to better understand these steps.

1. Register as BaseSpace developer. You can do this at developer.basespace.illumina.com/dashboard.
2. Understand the types of applications you can develop using BaseSpace and the underlying API data model. Section 3 gives an overview of the data model exposed by the BaseSpace API and the way is implemented in `BaseSpaceR`.
3. Register your new application using BaseSpace's application management. You will need the `client_id` and the `client_secret` to enable communication between your App and BaseSpace.
4. Learn the Authentication process and BaseSpace Permissions. Section 5 shows how manage these processes from within R.
5. Think about what data your application will need and which BaseSpace resources you'll have to query. Section 4 gives an overview on how to manipulate some of these resources.
6. Refer to the Release Notes to keep up on changes to the REST API and BaseSpace Developer Group to ask questions or provide feedback.

We further assume the user has some familiarity with the BaseSpace API and the R environment. Most of the exposed R methods take the same parameters as the REST methods they implement/use and we advise the developers to regularly check the REST API reference documentation.

2 Installation

This section briefly describes the necessary steps to get `BaseSpaceR` running on your system. We assume the user have the R environment already installed and it is fairly familiar with it. R 2.15.0 or later and `RCurl` and `RJSONIO` packages are required for installing and running `BaseSpaceR`.

The `BaseSpaceR` package is available from the Bioconductor repository. In order to install the package one needs first to install the core Bioconductor packages. If you have already installed Bioconductor packages on your system then you can skip the two lines below.

```
> if (!requireNamespace("BiocManager", quietly=TRUE))
+     install.packages("BiocManager")
> BiocManager::install()
```

Once the core Bioconductor packages are installed, we can install the `BaseSpaceR` package by:

```
> if (!requireNamespace("BiocManager", quietly=TRUE))
+     install.packages("BiocManager")
> BiocManager::install("BaseSpaceR")
```

To use the package we load it using the `library()` function.

```
> library(BaseSpaceR)
```

Note that when the `BaseSpaceR` package is loaded, both `RCurl` and `RJSONIO` packages are automatically loaded.

3 BaseSpace data model

The data model exposed by the BaseSpace API consists of the following major **resources/entities**:

Users: A user is the individual performing the analysis in BaseSpace.

Runs: A run is a collection of BCL files associated with an individual flow cell and contains metrics and reads generated by the instrument on which the flow cell was sequenced.

Projects: A project is a logical grouping of Samples and AppResults for a user.

Samples: A sample is a collection of FASTQ files of Reads. Sample contains metadata about the physical subject. Samples files are generally used as inputs to Apps.

AppResults: An AppResult is an output of an App. It contains BAMs, VCFs and other output file formats produced by an App.

AppSessions: AppSession extends information about each Sample and AppResult and allows grouping by showing the instance of an application.

Files: The files associated with a Run, Sample, or AppResult. All files for each resource are in the File resource.

Genomes: These are the reference genome sequences that exist in BaseSpace. This resource gives information about the origin and build for each genome

`BaseSpaceR` provides classes and methods to interface with this data model. Each **resource** is modeled by a S4 class. For every query the REST server receives, a response in the form of a JSON object is return.

Conceptually there are two response types exposed by BaseSpace, an individual response and a collection response. The former, modeled by the R `Item` class, is used when querying an individual item/instance (specified by the item ID) within a resource. The later is used for listing the items/instances available for a given resource and is modeled by the R `Collection` class.

3.1 `Item` objects

The `Item` class models a simple unordered set of key/value pairs. There is a core set of keys, for which access methods are defined and which are inherited by any child class. These methods are: `Id()`, `Name()`, `Href()`, `DateCreated()`, `UserOwnedBy()`, `Status()`, and `HrefBaseSpaceUI()`.

The class also implements a generic user level access operator, the `$` operator. It behavior of this operator is similar to its behavior on the `list` object. Thus, if the specified name is not a key of the instance, then NULL is returned. We can use this operator to access the core keys and we the following `Id(x) == x$Id` stands for all core keys.

3.2 `Collection` objects

The `Collection` class models an ordered set of `Item` objects and a set of predefined attributes. The interface provided by the `Item` class is implemented by this class. However, since we deal with an ordered set of objects, the methods and the access methods, return a vector of the same length as the size of the collection.

`Collection` class implements the following methods for accessing the set attributes: `Items()`, `DisplayedCount()`, `TotalCount()`, `Offset()`, `Limit()`, `SortDir`, and `SortBy()`.

This is in line with the way the REST API defines the response objects.

3.3 `Response` objects

Response objects consists of either an `Item` or a `Collection` object and an `AppAuth` object (the handler used to communicate with the server, see Section 5).

Every class representing a BaseSpace resource implements the `Response` interface. For example, for the Projects resource we have two classes `Projects` and `ProjectsSummary`. The former implements the `Item` interface, while the latter implements the `Collection` interface. Queries to the Projects resource return an instance one of these two objects

4 Sample Session - Quick API guide

This section describes a few of the API calls implemented by this SDK, that should be sufficient to give the user a flavor of how it works. Please note that within this document we are limited by the permission to the BaseSpace resources. Therefore the examples shown here are restricted to browsing the data.

A more comprehensive example, in which we select the FASTQ file(s) from a sample and compute Q-score statistics, can be seen in the other vignette shipped with this package, `BaseSpaceR-QscoreApp.pdf`.

A typical BaseSpace session can be divided into the following steps:

- Client authentication
- Data retrieval and access
- Data processing
- Uploading results back to BaseSpace

Bellow we'll focus on the first two steps and we'll briefly describe the others.

Authentication The first step is the authentication of the client application with BaseSpace. There are several ways in which this process can be triggered. Section 5 details the authentication process and the available options.

The communication between the client and the server is handled by an `AppAuth` instance. `BaseSpaceR` comes with an `AppAuth` instance offering restricted access to BaseSpace resources. For simplicity, we'll use this instance here.

```
> data(aAuth)
> aAuth

Object of class "AppAuth" with:

Client Id:
Client Secret:

Server
URL:      https://api.basespace.illumina.com
Version:  v1pre3

Authorized:      TRUE
```

If the access token is valid, which it is in our case, then the connection with the server is established and we can see it by printing the `AppAuth` object (Authorized: TRUE). Alternatively, we can use the `hasAccess` method, which returns `TRUE` if the access token is valid and `FALSE` otherwise.

Now that our client App is authenticated with the server we can start performing queries to the BaseSpace resources. xsa

4.1 Users

The Users resource allows the client to get basic information about the user that is currently using the application. To query this resource we use the `Users()` method.

```
> u <- Users(aAuth)
> u

Users object:
{
        "Email" : "illumina.basespacer@gmail.com",
        "HrefRuns" : "v1pre3/users/5638639/runs",
        "HrefProjects" : "v1pre3/users/5638639/projects",
        "Id" : "5638639",
        "Name" : "Illumina BaseSpaceR",
        "Href" : "v1pre3/users/5638639",
        "DateCreated" : "2015-04-21T10:36:19.0000000Z"
}
```

The response is an object of class `Users`. We can access the elements of this class using the generic access methods. For example, we can get the Id and the Name element as follows:

```
> Id(u)

[1] "5638639"

> Name(u)

[1] "Illumina BaseSpaceR"
```

Not all elements of a `Response` object have methods associated to them. There are only a core of elements for which methods exists, see Section 3. However every element of a `Response` object can be accessed using the general access operator `'$'` and the element key/name.

```
> u$Id
```

```
[1] "5638639"

> u$Email

[1] "illumina.basespacer@gmail.com"

> u$fakeElement

NULL
```

When an invalid key/name is given to the access method, then `NULL` is returned. The elements that belong to the object but have not been set can also be accessed and the method returns their default value.

```
> u <- Users()
> u

Users object:
{}

> u$Id

character(0)

> u$UserOwnedBy

list()
```

If the access token permits one can query for a particular user ID. The ID can be given either as an integer or as a string (this convention for specifying IDs holds for all API calls).

```
> Users(aAuth, id = 1463464)

 {
        "ResponseStatus" : {
                "ErrorCode" : "BASESPACE.USERS.USER_ID_NOT_VALID",
                "Message" : "Given user is not valid. Only 'current' user is allowed for this request
        },
        "Notifications" : [
                {
                        "Type" : "error",
                        "Item" : "Given user is not valid. Only 'current' user is allowed for this re
                }
        ]
}
NULL

> Users(aAuth, id = "1463464")

 {
        "ResponseStatus" : {
                "ErrorCode" : "BASESPACE.USERS.USER_ID_NOT_VALID",
                "Message" : "Given user is not valid. Only 'current' user is allowed for this request
        },
        "Notifications" : [
                {
```

```
                        "Type" : "error",
                        "Item" : "Given user is not valid. Only 'current' user is allowed for this re
                }
        ]
}
NULL
```

4.2 Genomes

The Genomes resource provides access the reference genomes available in BaseSpace. To access this resource we use `listGenomes()` and `Genomes()` methods.

`listGenomes()` lists all the available genomes, returning only a small summary for each genome. This is a general pattern across this API. Each resource implements a method that lists all entries/items visible to the current user, and a method which retrieves a particular item (specified via the item ID).

```
> g <- listGenomes(aAuth, Limit = 100)
> g$SpeciesName
```

```
 [1] "Arabidopsis thaliana"     "Bos Taurus"               "Escherichia coli"
 [4] "Homo sapiens"             "Mus musculus"             "Phix"
 [7] "Rhodobacter sphaeroides"  "Rattus norvegicus"        "Saccharomyces cerevisiae"
[10] "Staphylococcus aureus"    "Bacillus Cereus"          "Mus musculus"
[13] "Drosophila melanogaster"  "Escherichia coli"         "Rattus norvegicus"
```

Here we asked for at most 100 genomes available in BaseSpace. `Limit = 100` is a query parameter supported by the REST API. It tells the server that at most 100 items must be returned. If missing the dafault value of 10 is used.

`listGenomes()` returns a `GenomesSummary` object which basically contains a `Collection` object and an `AppAuth` handler. We can therefore use the `Collection` interface. We can see the number of items in the returned collection using `length()` or `DisplayCount` and the total number of items visible to the current user in BaseSpace using `TotalCount()`.

```
> length(g)
```

```
[1] 15
```

```
> TotalCount(g)
```

```
[1] 15
```

`Collection` objects implement the `[` and `[[` operators for sellection and subsetting. Thus if we want the 3rd item from the collection we do:

```
> g[[3]]
```

```
{
        "SpeciesName" : "Escherichia coli",
        "Source" : "NCBI",
        "Build" : "2008-03-17",
        "Id" : "3",
        "Href" : "v1pre3/genomes/3"
}
```

```
> is(g[[3]], "Item")
```

```
[1] TRUE
```

So the `[[` operator will always return an `Item` object. We can subset the collection, by giving an index:

```
> g[2:4]
```

```
GenomesSummary object:
Collection with 3 genomeItem objects (out of a total of 15 objects).
{
        "SpeciesName" : "Bos Taurus",
        "Source" : "Ensembl",
        "Build" : "UMD3.1",
        "Id" : "2",
        "Href" : "v1pre3/genomes/2"
}
{
        "SpeciesName" : "Escherichia coli",
        "Source" : "NCBI",
        "Build" : "2008-03-17",
        "Id" : "3",
        "Href" : "v1pre3/genomes/3"
}
{
        "SpeciesName" : "Homo sapiens",
        "Source" : "UCSC",
        "Build" : "hg19",
        "Id" : "4",
        "Href" : "v1pre3/genomes/4"
}
```

```
> g[1]
```

```
GenomesSummary object:
Collection with 1 genomeItem objects (out of a total of 15 objects).
{
        "SpeciesName" : "Arabidopsis thaliana",
        "Source" : "NCBI",
        "Build" : "build9.1",
        "Id" : "1",
        "Href" : "v1pre3/genomes/1"
}
```

The subsetting operator `[` works as it works for R vectors and lists. It returns an object of the same class of the original object.

In our case the collection size is 15, and since we specified `Limit = 100` in the call, we retrieve the complete collection. Please check the REST API documentation for the complete list of query parameters. We can see bellow how different parameters affect the response:

```
> listGenomes(aAuth, Limit = 2)
```

```
GenomesSummary object:
Collection with 2 genomeItem objects (out of a total of 15 objects).
{
        "SpeciesName" : "Arabidopsis thaliana",
        "Source" : "NCBI",
        "Build" : "build9.1",
        "Id" : "1",
```

```
        "Href" : "v1pre3/genomes/1"
}
{
        "SpeciesName" : "Bos Taurus",
        "Source" : "Ensembl",
        "Build" : "UMD3.1",
        "Id" : "2",
        "Href" : "v1pre3/genomes/2"
}
```

```
> g <- listGenomes(aAuth, Offset = 5, Limit = 2, SortBy = "Build")
> g
```

```
GenomesSummary object:
Collection with 2 genomeItem objects (out of a total of 15 objects).
{
        "SpeciesName" : "Arabidopsis thaliana",
        "Source" : "NCBI",
        "Build" : "build9.1",
        "Id" : "1",
        "Href" : "v1pre3/genomes/1"
}
{
        "SpeciesName" : "Drosophila melanogaster",
        "Source" : "UCSC",
        "Build" : "dm3",
        "Id" : "2003",
        "Href" : "v1pre3/genomes/2003"
}
```

```
> TotalCount(g) # Collection size remains constant
```

```
[1] 15
```

We can to access all the information available for a specific genome using the `Genomes()` method. Let's assume we want to do this for the "Homo sapiens" genome, which has ID 4.

```
> Genomes(aAuth, id = 4)
```

```
$`4`
Genomes object:
{
        "SpeciesName" : "Homo sapiens",
        "Source" : "UCSC",
        "Build" : "hg19",
        "Id" : "4",
        "Href" : "v1pre3/genomes/4"
}
```

Multiple IDs can be specified. In this case the returned value is a named list, with the IDs given the list' names. If a given IDs is not valid, then the respective element in the list is set to `NULL`. At the same the server error message is shown.

```
> Genomes(aAuth, id = c(4, 1, 110))
```

```
 {
        "ResponseStatus" : {
```

```
                "ErrorCode" : "NotFound",
                "Message" : "Genome id '110' doesn't exist"
        },
        "Notifications" : [
                {
                        "Type" : "error",
                        "Item" : "Genome id '110' doesn't exist"
                }
        ]
}
$`4`
Genomes object:
{
        "SpeciesName" : "Homo sapiens",
        "Source" : "UCSC",
        "Build" : "hg19",
        "Id" : "4",
        "Href" : "v1pre3/genomes/4"
}

$`1`
Genomes object:
{
        "SpeciesName" : "Arabidopsis thaliana",
        "Source" : "NCBI",
        "Build" : "build9.1",
        "Id" : "1",
        "Href" : "v1pre3/genomes/1"
}

$`110`
NULL
```

Please note that if `id` has length 1, a list with one element is returned. By specifying `simplify = TRUE`, the list is dropped, and a `Genomes` object is returned.

```
> Genomes(aAuth, id = 4, simplify = TRUE)

Genomes object:
{
        "SpeciesName" : "Homo sapiens",
        "Source" : "UCSC",
        "Build" : "hg19",
        "Id" : "4",
        "Href" : "v1pre3/genomes/4"
}
```

We can use the `Genomes()` method on a `GenomesSummary` object. In this case a `Genomes` object will be returned for every item in collection.

```
> Genomes(g)

$`1`
Genomes object:
{
        "SpeciesName" : "Arabidopsis thaliana",
        "Source" : "NCBI",
        "Build" : "build9.1",
```

```
        "Id" : "1",
        "Href" : "v1pre3/genomes/1"
}

$`2003`
Genomes object:
{
        "SpeciesName" : "Drosophila melanogaster",
        "Source" : "UCSC",
        "Build" : "dm3",
        "Id" : "2003",
        "Href" : "v1pre3/genomes/2003"
}
```

4.3 Runs

The Runs resource contains the raw data produced by the instruments, the base calls, together with run metrics, instrument health data, and other information used for data processing and analysis.

To list the available runs we use `listRuns()` method.

```
> r <- listRuns(aAuth)
> r

RunsSummary object:
Collection with 2 runItem objects (out of a total of 2 objects).
{
        "UserUploadedBy" : {
                "Id" : "1001",
                "Href" : "v1pre3/users/1001",
                "Name" : "Illumina Inc",
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7aaea7cc
        },
        "ExperimentName" : "BacillusCereus",
        "Id" : "101102",
        "Name" : "110513_M10_0089_AA0172",
        "Href" : "v1pre3/runs/101102",
        "DateCreated" : "2012-01-14T00:10:25.0000000Z",
        "UserOwnedBy" : {
                "Id" : "1001",
                "Href" : "v1pre3/users/1001",
                "Name" : "Illumina Inc",
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7aaea7cc
        },
        "Status" : "Failed",
        "HrefBaseSpaceUI" : "https://basespace.illumina.com/run/101102/BacillusCereus"
}
{
        "UserUploadedBy" : {
                "Id" : "1001",
                "Href" : "v1pre3/users/1001",
                "Name" : "Illumina Inc",
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7aaea7cc
        },
        "ExperimentName" : "2x151PhiX",
        "Id" : "277282",
```

```
        "Name" : "11-09-15_M11_0199_A-RG1234567-00CDE",
        "Href" : "v1pre3/runs/277282",
        "DateCreated" : "2012-04-25T15:19:25.0000000Z",
        "UserOwnedBy" : {
                "Id" : "1001",
                "Href" : "v1pre3/users/1001",
                "Name" : "Illumina Inc",
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7aaea7cc
        },
        "Status" : "Complete",
        "HrefBaseSpaceUI" : "https://basespace.illumina.com/run/277282/2x151PhiX"
}
```

And as before, we can use various query parameters. For example, we can ask for all runs that failed.

```
> listRuns(aAuth, Statuses = "Failed") # no failed runs in our case

RunsSummary object:
Collection with 1 runItem objects (out of a total of 1 objects).
{
        "UserUploadedBy" : {
                "Id" : "1001",
                "Href" : "v1pre3/users/1001",
                "Name" : "Illumina Inc",
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7aaea7cc
        },
        "ExperimentName" : "BacillusCereus",
        "Id" : "101102",
        "Name" : "110513_M10_0089_AA0172",
        "Href" : "v1pre3/runs/101102",
        "DateCreated" : "2012-01-14T00:10:25.0000000Z",
        "UserOwnedBy" : {
                "Id" : "1001",
                "Href" : "v1pre3/users/1001",
                "Name" : "Illumina Inc",
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7aaea7cc
        },
        "Status" : "Failed",
        "HrefBaseSpaceUI" : "https://basespace.illumina.com/run/101102/BacillusCereus"
}
```

If we are interested in one or more runs, we use the method with the same name as the resource, `Runs()` in this case.

```
> myRun <- Runs(r[1], simplify = TRUE)
> myRun

Runs object:
{
        "HrefFiles" : "v1pre3/runs/101102/files",
        "HrefSamples" : "v1pre3/runs/101102/samples",
        "UserUploadedBy" : {
                "Id" : "1001",
                "Href" : "v1pre3/users/1001",
                "Name" : "Illumina Inc",
```

```
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7aaea7cc
        },
        "DateUploadCompleted" : "2012-01-14T01:02:07.0000000Z",
        "DateUploadStarted" : "2012-01-14T00:10:25.0000000Z",
        "ExperimentName" : "BacillusCereus",
        "Id" : "101102",
        "Name" : "110513_M10_0089_AA0172",
        "Href" : "v1pre3/runs/101102",
        "DateCreated" : "2012-01-14T00:10:25.0000000Z",
        "UserOwnedBy" : {
                "Id" : "1001",
                "Href" : "v1pre3/users/1001",
                "Name" : "Illumina Inc",
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7aaea7cc
        },
        "Status" : "Failed",
        "HrefBaseSpaceUI" : "https://basespace.illumina.com/run/101102/BacillusCereus"
}
```

Runs have files associated with them. We can list the files associated with the selected run using the `listFiles()` method. More details on the `Files` object and resource are given in section 4.5.

```
> f <- listFiles(myRun)
> Name(f)

 [1] "runparameters.xml"          "runinfo.xml"                "samplesheet.csv"
 [4] "queuedforanalysis.txt"      "rtacomplete.txt"            "extractionmetricsout.bin"
 [7] "tilemetricsout.bin"         "correctedintmetricsout.bin" "qmetricsout.bin"
[10] "controlmetricsout.bin"
```

We can chose to select a particular type of files, base on the file extension. Here we want to select two BCL files.

```
> listFiles(myRun, Limit = 2, Extensions = ".bcl")

FilesSummary object:
Collection with 2 fileItem objects (out of a total of 3696 objects).
{
        "HrefContent" : "v1pre3/files/r101102_529844/content",
        "Size" : 548794,
        "Path" : "data/intensities/basecalls/l001/c171.1/s_1_5.bcl",
        "ContentType" : "application/octet-stream",
        "Id" : "r101102_529844",
        "Name" : "s_1_5.bcl",
        "Href" : "v1pre3/files/r101102_529844",
        "DateCreated" : "2012-01-14T00:10:30.0000000Z"
}
{
        "HrefContent" : "v1pre3/files/r101102_529845/content",
        "Size" : 527125,
        "Path" : "data/intensities/basecalls/l001/c171.1/s_1_11.bcl",
        "ContentType" : "application/octet-stream",
        "Id" : "r101102_529845",
        "Name" : "s_1_11.bcl",
        "Href" : "v1pre3/files/r101102_529845",
        "DateCreated" : "2012-01-14T00:10:30.0000000Z"
}
```

4.4 Projects

The Project resource provides a logical grouping of the Samples resource and the AppResults resource for a given user.

Accessing the Projects resource is similar as for any most resources. There is a `listProjects()` method for browsing the available projects and the `Projects()` method for selecting and retrieving project data.

```
> Projects(listProjects(aAuth, Limit = 1), simplify = TRUE)

Projects object:
{
        "HrefSamples" : "v1pre3/projects/21383369/samples",
        "HrefAppResults" : "v1pre3/projects/21383369/appresults",
        "Id" : "21383369",
        "Name" : "MiSeq: TruSeq 2x151 (PhiX)",
        "Href" : "v1pre3/projects/21383369",
        "DateCreated" : "2015-03-31T07:11:09.0000000Z",
        "UserOwnedBy" : {
                "Id" : "1001",
                "Href" : "v1pre3/users/1001",
                "Name" : "Illumina Inc",
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7aaea7cc
        },
        "HrefBaseSpaceUI" : "https://basespace.illumina.com/project/21383369/MiSeq-TruSeq-2x151-PhiX"
}
```

Unlike the resources seen until now, for Projects, the users may create new and/or share existing projects. We can add a new project using the `createProject()` method.

```
> myNewProj <- createProject(aAuth, name = "My New Project")

Project 'My New Project' successfully created. Assigned Id: 22011990

> myNewProj

Projects object:
{
        "HrefSamples" : "v1pre3/projects/22011990/samples",
        "HrefAppResults" : "v1pre3/projects/22011990/appresults",
        "Id" : "22011990",
        "Name" : "My New Project",
        "Href" : "v1pre3/projects/22011990",
        "DateCreated" : "2015-04-22T09:52:03.0000000Z",
        "UserOwnedBy" : {
                "Id" : "5638639",
                "Href" : "v1pre3/users/5638639",
                "Name" : "Illumina BaseSpaceR",
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/1f0003cc4474154aec01ca696bffded8
        },
        "HrefBaseSpaceUI" : "https://basespace.illumina.com/project/22011990/My-New-Project"
}
```

As we can see the function failed to create a new project. This is because one needs additional rights, in this case the user needs to grant `create project` access, to be able to create a new resource in BaseSpace. See BaseSpace Permissions for more details on the permission structure in BaseSpace, and Section 5 for details on renewing permissions.

Samples and AppResults The Samples and AppResults resources have similar interfaces. There are no files directly associated with a Project, but both Samples and AppResults each have files within them.

In general, samples are the result of demultiplexing and can be considered to be holding the input data for an App. One example of data within a Samples resource are the FASTQ files.

The AppResults resource is used for keeping the result of an App. On top of the typical listing and accessing data from this resource, the user can also create new and update existing AppResults. The method used for adding a new AppResult instance is `createAppResults()`. As in the case of Projects, the user need the right permission to be able to create a new instance.

We can quickly browse an AppResults from the project with ID 21383369.

```
> reseq <- listAppResults(aAuth, projectId = 21383369, Limit = 1)
> AppResults(reseq)

AppResults object:
{
        "HrefFiles" : "v1pre3/appresults/22137373/files",
        "AppSession" : {
                "Id" : "24372411",
                "Name" : "MSR: Resequencing",
                "Href" : "v1pre3/appsessions/24372411",
                "Application" : {
                        "Id" : "2938944",
                        "Href" : "v1pre3/applications/2938944",
                        "Name" : "MSR: Resequencing",
                        "CompanyName" : "Illumina, Inc.",
                        "VersionNumber" : "1.0.0",
                        "HomepageUri" : "www.illumina.com",
                        "ShortDescription" : "resequencing",
                        "DateCreated" : "2015-03-28T23:33:16.0000000Z",
                        "PublishStatus" : "Development",
                        "IsBillingActivated" : false,
                        "Category" : "Analytic",
                        "Classifications" : "Illumina",
                        "AppFamilySlug" : "illumina-inc.msr-resequencing",
                        "AppVersionSlug" : "illumina-inc.msr-resequencing.1.0.0",
                        "Features" : "IsAppRegisteredToPlatform",
                        "LockStatus" : "Unlocked"
                },
                "UserCreatedBy" : {
                        "Id" : "1001",
                        "Href" : "v1pre3/users/1001",
                        "Name" : "Illumina Inc",
                        "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                        "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7
                },
                "Status" : "Complete",
                "QcStatus" : "Undefined",
                "StatusSummary" : "",
                "DateCreated" : "2015-03-31T06:43:03.0000000Z",
                "ModifiedOn" : "2022-12-14T18:15:31.0000000Z",
                "DateCompleted" : "2015-03-31T07:11:09.0000000Z",
                "TotalSize" : 3.937933e+09
        },
        "Id" : "22137373",
        "Name" : "Phix",
        "Href" : "v1pre3/appresults/22137373",
        "DateCreated" : "2015-03-31T07:11:09.0000000Z",
        "UserOwnedBy" : {
```

```
                "Id" : "1001",
                "Href" : "v1pre3/users/1001",
                "Name" : "Illumina Inc",
                "DateCreated" : "0001-01-01T00:00:00.0000000Z",
                "GravatarUrl" : "https://secure.gravatar.com/avatar/0646200388a465f27694d67b7aaea7cc
        },
        "Status" : "Complete"
}
```

We can see that there are references to the files and genomes associated with this resequencing result.

A more comprehensive examples, showing how to use these resources can be found in the `BaseSpaceR-QscoreApp.pdf` vignette, which one can find here:

```
> system.file("doc", "BaseSpaceR-QscoreApp.pdf", package = "BaseSpaceR")
```

4.5 Files

The Files resource provides access to files stored in BaseSpace. A file should be seen as a data stream and associated attributes (date created, size, type, etc.).

Files are associated with specific Runs, Samples, or AppResults and the Files resource provides the interface for manipulating these files. For each one of the above mentioned resource, one can list the available files using the `listFiles()` method.

The method can be called using an instance of `Runs`, `Samples`, or `AppResults`, in which case the files for the respective resource will be listed, or using an `AppAuth` handler, in which case the ID of the wanted resource must be specified. To list the files associated with the resequncing result selected in section 4.4 we do:

```
> f <- listFiles(AppResults(reseq))
> TotalCount(f)
```

```
[1] 24
```

```
> Name(f)
```

```
 [1] "Phix_S1.vcf.index"                          "CompletedJobInfo.xml"
 [3] "ResequencingRunStatistics.1.xml"            "ResequencingRunStatistics.xml"
 [5] "CompletedJobInfo.xml"                       "OverallByCycle.daf"
 [7] "s_G1.1.phix.ByPosition.daf"                 "ErrorsAndNoCallsByLaneTileReadCycle.csv"
 [9] "tmp.segment.tempvariants_G1.1.0.vcf.stdout" "Mismatch.htm"
```

If we knew only the ID of the AppResults instance, then the call could have been done as follows:

```
> identical(f, listFiles(aAuth, appResultId = Id(reseq)))
```

```
[1] TRUE
```

To specify the ID for Runs and Samples use `runID = <ID>` and `sampleId = <ID>` respectively.

Specific file types can be selected base on the file extension. Let's assume we are interested in BAM files.

```
> f <- listFiles(aAuth, appResultId = Id(reseq), Extensions = ".bam")
> Name(f)
```

```
[1] "Phix_S1.bam"
```

Detailed information about files are available via the `Files` method. One can call the function using a `FilesSummary` object (the object returned by `listFiles()`) or using the `AppAuth` handler and the file ID.

```
> Files(f)

$`1668718877`
Files object:
{
        "UploadStatus" : "complete",
        "HrefContent" : "v1pre3/files/1668718877/content",
        "HrefCoverage" : "v1pre3/coverage/1668718877",
        "Size" : 1017123411,
        "Path" : "data/intensities/basecalls/Alignment/Phix_S1.bam",
        "ContentType" : "application/octet-stream",
        "Id" : "1668718877",
        "Name" : "Phix_S1.bam",
        "Href" : "v1pre3/files/1668718877",
        "DateCreated" : "2015-03-31T07:11:09.0000000Z"
}
```

Access to the files attributes and the data stream is restricted and requires additional permissions. The equivalent of the above call, for an `AppAuth` handler is: `Files(aAuth, id = Id(f))`. Please note that `id` is the ID of a file regardless of the resource it belongs to.

There is a much richer API for the Files resource. One can download the selected files, upload a file and set the files attributes. Multiple file uploads are also supported. However all these operations require additional permissions and are out of the scope of this document. Please see `BaseSpaceR-QscoreApp.pdf` for more elaborate examples using files.

Coverage and Variants Certain files, in particular VCF and BAM files, have additional information available. BaseSpace offers to separate resources for these two types, Coverage and Variants.

The Coverage resource is used to provide mean read coverage depth in a particular chromosomal region. Querying the resources we get the coverage depth histogram. This resource is at this moment implemented only for BAM files. We can find the BAM files in an AppResults instance.

```
> bamFiles <- listFiles(AppResults(reseq), Extensions = ".bam")
> Name(bamFiles)

[1] "Phix_S1.bam"

> Id(bamFiles)

[1] "1668718877"

> bamFiles

FilesSummary object:
Collection with 1 fileItem objects (out of a total of 1 objects).
{
        "HrefContent" : "v1pre3/files/1668718877/content",
        "Size" : 1017123411,
        "Path" : "data/intensities/basecalls/Alignment/Phix_S1.bam",
        "ContentType" : "application/octet-stream",
        "Id" : "1668718877",
        "Name" : "Phix_S1.bam",
        "Href" : "v1pre3/files/1668718877",
        "DateCreated" : "2015-03-31T07:11:09.0000000Z"
}
```

There are two functions implemented for this resource, `getCoverageStats()` and `getCoverage()`.

```
> getCoverageStats(aAuth, id = Id(bamFiles), "phix")
```

```
[[1]]
[[1]]$MaxCoverage
[1] 332136

[[1]]$CoverageGranularity
[1] 128
```

Unfortunately, the user needs to grant read access to the selected AppResults instance, to be able to access the raw BAM data.

The Variants resource provides access to variants within a VCF file. It allows for efficient querying the variants of a particular chromosomal region. The R API functions to interact with this resource are `getVariantSet()` and `getVariants()`. As with the Coverage resource, one needs additional permission to access this resource.

To find VCF files, we query the AppResults resource and select files base on the file extension.

```
> vcfs <- listFiles(AppResults(reseq), Extensions = ".vcf")
> Name(vcfs)
```

```
[1] "Phix_S1.vcf"
```

```
> Id(vcfs)
```

```
[1] "1668718881"
```

```
> vcfs
```

```
FilesSummary object:
Collection with 1 fileItem objects (out of a total of 1 objects).
{
        "HrefContent" : "v1pre3/files/1668718881/content",
        "Size" : 5816,
        "Path" : "data/intensities/basecalls/Alignment/Phix_S1.vcf",
        "ContentType" : "text/vcard",
        "Id" : "1668718881",
        "Name" : "Phix_S1.vcf",
        "Href" : "v1pre3/files/1668718881",
        "DateCreated" : "2015-03-31T07:11:09.0000000Z"
}
```

And as above, we are not able to access this resource, give the restricted access token we are using throughout this document.

```
> getVariants(aAuth, Id(vcfs)[1], chrom = "chr", EndPos = 1000000L, Limit = 5)
```

5 Authentication

The authentication and communication between the client, the R application, and the server, the BaseSpace REST server, is handled by an object of class `AppAuth`.

It is easy to create an `AppAuth` instance, and there a few options for doing this. However, before going further and detail the interface this object provides, we need to clarify the authentication process. First we advise developers to familiarize with BaseSpace authentication and the OAuth v2 workflow.

To start the authentication process the developer must have the `client_id` and the `client_secret`. These are available in BaseSpace's application management screen of your app. If this is the first time you hear about `client_id` and `client_secret` please read through Steps for developing your first App section before going forward with this document.

BaseSpaceR offers the following options for an App to authenticate with BaseSpace.

- Use a pre-generated access token.
- Use the OAuth v2 process. This is the recommended option, but this requires the user interacting directly with a Web browser.
- Use the AppSessions and BaseSpace Application Triggering workflow. However this requires a web server to process App launch. *[under development]*

5.1 The OAuth v2 workflow

To start the authentication process we need to instantiate an `AppAuth` object with the `client_id` and the `client_secret`.

```
> myAppClientId <- "aaaaa8acb37a441fa71af5072fd7432b"
> myAppClientSecret <- "bbbbb8acb37a441fa71af5072fd7432b"
> aAuth <- AppAuth(client_id = myAppClientId,
+                  client_secret = myAppClientSecret,
+                  scope = "create global")
```

The above code will show a server error, which is expected since the provided `client_id` and `client_secret` are fake. However, the `AppAuth` instance is valid, just not authenticated with the server.

Printing the handler will show basic information. Note that since there is no valid access token associated with this object we get *Authorized: FALSE*.

The permission type is specified using the `scope` parameter. See BaseSpace Permissions for a comprehensive description on how to define the `scope`. If `scope` parameter is missing the default setting, `browse global` is used.

If the specified `client_id` and `client_secret` were valid, the `AppAuth()` constructor would have initiated the authentication process, in which case the user is shown the following message:

```
Perform OAuth authentication using the following URI:
    https://basespace.illumina.com/oauth/device?code=xxxxx
```

The App user needs to open the URI in a browser and perform the authentication. The app will have to idle until further user input.

Alternatively one can instantiate an `AppAuth` object without triggering the authentication process. This is achieved by specifying `doOAuth = FALSE` in the function call.

```
> aAuth <- AppAuth(client_id = myAppClientId,
+                  client_secret = myAppClientSecret,
+                  scope = "read global",
+                  doOAuth = FALSE)
> aAuth

Object of class "AppAuth" with:
```

```
Client Id:       aaaaa8acb37a441fa71af5072fd7432b
Client Secret:  bbbbb8acb37a441fa71af5072fd7432b

Server
URL:      https://api.basespace.illumina.com
Version:  v1pre3

Authorized:      FALSE
```

The OAuth authentication process can be triggered at any point using the `initializeAuth()` method. We can also specify a new scope when calling the function. In this case, the previous scope associated with the `aAuth` handler will be updated. For example, an empty scope, allow us to access the user information.

```
> res <- initializeAuth(aAuth, scope = character())
> res

[1] 401
```

The function returns a list with two entries. The URI and the device code. This can be used by the developer to automatically launch a browser, or create custom messages for the user.

Obtaining an Access Token Once the user has granted access using the OAtuh v2 authentication dialog box, we can request the access token from the server and update the `aAuth` handler. This is achieved by calling the `requestAccessToken()` function.

```
> requestAccessToken(aAuth)
> hasAccess(aAuth)
```

If the request is successful, the following message is shown *Access token successfully acquired!*. Calling `hasAccess()` or printing the `AppAuth` instance will show 'Authorized: TRUE'. If the request fails, the response returned by the REST API is show. For example, if the user didn't authorized the request we get the following message:

```
BadRequest
{
  "error" : "authorization_pending",
  "error_description" : "User has not yet approved the access request"
}
```

5.2 Using a pre-generated access token

One can instantiate an `AppAuth` object using a pre-generated access token. To pre-generate an access token we can use the above described OAuth v2 workflow and save the access token for further use, or use other tools available in BaseSpace.

A pre-generated access token with restricted permissions can be obtained from the `aAuth` data.

```
> data(aAuth)
> app_access_token <- aAuth$access_token
```

`app_access_token` contains the 32 character string access token. We can now instantiate a new handler as follows:

```
> newAuth <- AppAuth(access_token = app_access_token)
> newAuth
```

```
Object of class "AppAuth" with:

Client Id:
Client Secret:

Server
URL:      https://api.basespace.illumina.com
Version:  v1pre3

Authorized:      TRUE
```

If the access token is valid, which should be in our case, the connection with the server is established and we can see it by printing the `AppAuth` object *Authorized: TRUE*.

Please note that it does not make sense to specify a scope because the scope is already encapsulated in the access token. Also, the `client_id` and the `client_secret` are not required in this case, given that once the access token is available they are not needed to communicate with the server. However, they can be specified together with the pre-generated access token.

```
> newAuth <- AppAuth(access_token = app_access_token,
+                    client_secret = myAppClientSecret,
+                    client_id = myAppClientId)
> newAuth
```

```
Object of class "AppAuth" with:

Client Id:       aaaaa8acb37a441fa71af5072fd7432b
Client Secret:  bbbbb8acb37a441fa71af5072fd7432b

Server
URL:      https://api.basespace.illumina.com
Version:  v1pre3

Authorized:      TRUE
```

The advantage of having the `client_id` and the `client_secret` is that we can re-initiate the OAuth v2 process at any point, and update the handler scope.

6 Session Information

The version number of R and packages loaded for generating the vignette were:

- R Under development (unstable) (2025-10-20 r88955), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_GB`, `LC_COLLATE=C`, `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=en_US.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Time zone: `America/New_York`
- TZcode source: `system (glibc)`
- Running under: `Ubuntu 24.04.3 LTS`
- Matrix products: default
- BLAS: `/home/biocbuild/bbs-3.23-bioc/R/lib/libRblas.so`
- LAPACK: `/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0`
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: BaseSpaceR 1.53.0, RCurl 1.98-1.17, RJSONIO 2.0.0
- Loaded via a namespace (and not attached): bitops 1.0-9, compiler 4.6.0, tools 4.6.0