# bgdata Documentation

*Release 2.0.0*

**BBGLab**

**Dec 07, 2018**

# Contents:

**bgdata** is a simple *data package* manager. It allows to create, search and use *data packages*.

By default, it works with the data packages used by our group: Barcelona Biomedical Genomics Group.

**bgdata** downloads once the packages from a remote *repository* and keeps them in a local repository (typically a local folder) so that it is fast to access them. To get a better overview of how the repositories work check the *repositories* section.

However, **bgdata** is more than that, keep reading to find out what it can do.

CHAPTER 1

# The data packages

A data package is nothing more, and nothing less, that set of files (or even a single file).

## 1.1 Identifying

**bgdata** identifies each data package with a 4-level structure

1. *project*
2. *dataset*
3. *version*
4. *build*

`project` and `dataset` are the *main identifiers* of a data package. In some cases, you might find that the package does not belong to a particular project. For such cases, we use _ as project name. Some of the **bgdata** commands will automatically set the project as _ if you do not provide it.

The `version` is intended distinguish between *incompatible* versions of the package. E.g. when you are removing some data columns in your files.

The `build` is an identifier that allows to distinguish between *compatible* versions of the same packages. Typically, we use the date when we create the package as the `build` identifier. However, the `build` can be anything (as long as it does not start with an alpha character), so you might find other builds.

For example, we use the human genome in many of projects. There are several version of the human genome available at http://hgdownload.cse.ucsc.edu/downloads.html#human . We downloaded our data of interest for the hg19 version and created packages using _ as *project*, `genomereference` as *dataset*, `hg19` as *version* and `20150724` as *build*. Then, we can request this package as

```
bgdata get _/genomereference/hg19?20150724
```

## 1.2 Tags

As remembering all the `build` identifiers for all the packages might be painful and you probably need to change all the queries in your scripts to get newer versions, **bgdata** supports the concept of **tags**.

A `tag` is a pointer to a particular build, and in several operations with **bgdata** you can use a `tag` instead of a `build`. **bgdata** will resolve which is the `build` associated with that `tag` and use that package.

The advantage of using a `tag` rather than a `build` is that with the same query in your software, you get the most updated version of a particular package by only keeping the `tag` up to date. E.g. following the example above, if we ask for the `tag` *master* we get the always the most recent version:

```
bgdata get _/genomereference/hg19?master
```

provided that we keep our `tag` up to date.

In most cases, **bgdata** will use the `tag` *master* when you do not indicate the `build` or `tag` for a particular package.

---

**Important:** A `tag` works essentially as a pointer to a `build` for a particular `project`, `dataset` and `version`. This means that when asking for a `tag` you also need to indicate the other parameters.

---

CHAPTER 2

# Repositories

**bgdata** manages the packages through 3 layers of repositories:

- *remote*
- *local*
- *caches*

## 2.1 Remote

The *remote* represents a repository that serves as a source of data packages. Currently, it is an HTTP server that contains the compressed data packages and some tags.

When the user requests for a package that is not present in the *local* repository **bgdata** will download it from the *remote* into the local.

In addition, **bgdata** will keep in sync the `tags`. This means that if a `tag` of a particular package is updated in the *remote*, and the user requests that particular `tag`, he or she will get the latest version from the *remote* if the local `tag` was not up to date.

**Note:** **bgdata** can work in offline mode. In such case, packages will not be downloaded and tags will not be updated.

## 2.2 Local

The local repository is the one where the user can find the packages that have been requested.

While the remote is an HTTP server, the local should be a reachable path from the user's machine.

The main difference with the *remote* repository, apart from being in the local machine, is that packages are uncompressed.

### 2.2.1 The download process

The download process from the remote is done using the Python package homura. Thanks to it, downloads can be resumed. After download, **bgdata** extracts all the files if they were compressed.

Once the download and extraction processes are done **bgdata** creates a file named `.download` with the date and time of that moment. If this file is not present or deleted, **bgdata** assumes the download has failed and reattempts it.

## 2.3 Caches

A *cache* is an extension of the local repository. Like the local repository, it should be reachable path from the user's machine. Moreover, **bgdata** supports multiple caches.

When the user request a packages, **bgdata** will be search for it first in each *cache* and the in the *local* repository.

A *cache* can have different uses. As an example, we use the scratch space in the nodes of our cluster to as cache for the packages we use recurrently. For the others, we have a *local* repository reachable through the network file system.

---

**Important:** **bgdata** will not fail just because a *cache* is not present. This means that you can also use an external hard drive as a *cache* and if it is not connected **bgdata** can still be used.

---

# CHAPTER 3

# Configuring bgdata

**bgdata** has a default configuration file which looks like:

```
version=2
local_repository = "~/.bgdata"
remote_repository = "http://bbglab.irbbarcelona.org/bgdata"
```

However, you can create you own configuration file and change it.

## 3.1 Custom configuration

To create you own custom configuration you need to create a file `bgdatav2.conf` and place in the corresponding config file folder (this is done using the appdir package using the `user_config_dir` function with `bbglab` as the only parameter).

That file, should follow the same structure as the default, but you can change the sections to fit you own needs.

---

The **local folder** (where the data packages are stored) is indicated through `local_repository`.

```
# The default local folder where you want to store the data packages
local_repository = "~/.bgdata"
```

---

**Note:** You can put any reachable path.

---

The **remote repository** is a (public) URL where the data packages are stored and the **bgdata** uses to look for the packages that are not in the local repository.

```
# The remote URL from where do you want to download the data packages
remote_repository = "http://bbglab.irbbarcelona.org/bgdata"
```

If you need to access to the remote repo through a proxy you can also configure it as follows:

```
# Optional proxy configuration
# [proxy]
host = proxy.someurl.org
port = 8080

# If it's an authenticated proxy
user = myname
pass = mypasswd
```

Optionally, **bgdata** can be set to **not** look for newer versions of the packages in the remote repository and only use what is available on the local. To make use of this option, you need to add:

```
# If you want to force bgdata to work only locally
offline = True
```

Using the `cache_repositories` option you can indicate a list of repositories (similar to the local) in which to look for the files.

```
# Cache repositories
[cache_repositories]
# Pairs name and path
my hard drive = /mnt/user/hd
```

**Note:** cache repositories have higher priority than the local, meaning that **bgdata** will look in them before checking the local. In addition, they are search last to first.

As an example of usage, data packages that are being used recurrently in our cluster are saved in the `scratch` directory of each node. This way, **bgdata** takes the data from the `scratch` which is faster than using the network file system.

CHAPTER 4

# Usage

**bgdata** is a Python package with a command line interface. This means that you can use **bgdata** as a Python library or from a terminal.

## 4.1 Getting packages

The most basic function of **bgdata** is to retrieve the **path** to a particular package. This is done through the **get** method.

The package is identified by a string with the format:

```
[<project>/]<dataset>/<version>[?<build>|<tag>]
```

- *project* is optional. Default project is _
- *dataset* and *version* are required
- *build* or *tag* are optional. By default, **bgdata** requests the `tag` **master**.

**Note:** As **master** is the default `tag`, it is present in the *remote* repository, and unless you are in *offline* mode, **bgdata** will keep it synchronized.

As an example, we are going to ask for **master** `tag` of *hg19* `version` the *genomereference* `dataset` in the default `project` (_).

From the command line:

```
$ bgdata get _/genomereference/hg19?master
2018-03-19 10:56:08 bgdata.manager INFO -- "master" resolved as 20150724
2018-03-19 10:56:08 bgdata.command INFO -- Dataset downloaded
/home/user/.bgdata/_/genomereference/hg19-20150724
```

and from Python:

```
>>> bgdata.get('_/genomereference/hg19?master')
'/home/user/.bgdata/_/genomereference/hg19-20150724'
```

**Important:** **bgdata** returns the path to local or cache folder where the package is present. When there is only one file in the folder, or in some special cases, **bgdata** returns the path to that file instead of the folder path.

## 4.2 Searching for packages

The **bgdata list** command can be used to check which data packages are in the local repository. This function (actually it is a generator) returns three elements: a string that represents the *package* (like the input for the *get* method), the *name of the repo* where you can find the package (`local` represents the local repository, and the rest will be the names of the caches), and the *tags* associated with that particular build.

In the command line:

```
$ bgdata list
_/genomereference/hg19?20150724      local    ['master']
```

From Python:

```
>>> for pkg, repo, tag in bgdata.list():
...     print('Package {} in {} is associated with tags: {}'.format(pkg, repo, tag))
...
Package _/genomereference/hg19?20150724 in local is associated with tags: ['master']
```

To search for packages you can use the **search** command. This command lists all available packages in the indicated level. For example, when searching with empty string, it will list all available *projects*:

```
$ bgdata search
_
cgi
intogen
```

If you search for a *project*, you get a list of *datasets*:

```
$ bgdata search _
genomereference
genomesignature
```

If you search for a *dataset* within a *project*, you get all possible *versions*:

```
$ bgdata search _/genomereference
hg19
hg18
hg38
```

And *builds* can be find out by searching for the *version* of the *dataset* within a *project*:

```
$ bgdata search _/genomereference/hg19
20150724
```

## 4.3 Informartion about the packages

The remote repository contains metadata about the packages. This information is used internally by **bgdata** to know which projects are presents, which datasets are in each project and so on.

The **info** command can be used to retrieve that information, by a simple query.

```
$ bgdata info _/genomereference/hg19
{'author': 'BBGLab',
 'created_on': '20150724,
 'description': 'Human Genome HG19',
 'license': 'Freely available for public use',
 'md5': '851d41ac755f4deba7b98851084927ab'
 'source': 'http://hgdownload.soe.ucsc.edu/goldenPath/hg19/chromosomes/'}
```

## 4.4 Logs

The logging process of **bgdata** is done using the `logging` module in Python.

When using **bgdata** as a Python library, the logging module is not configured at all, thus it is left to the end user how to configure the logging system. The loggers used by **bgdata** are all below one named as `bgdata` so you only need to configure that one.

When using **bgdata** from the command line interface, there are two flags that can be used to configure the logging system.

**bgdata** contains a set of subcommands but there are two flags that are general:

| | |
|---|---|
| **-v, --verbose** | Give more information |
| **-q, --quiet** | Suppress all log messages but the ones on the stderr |

The `--quiet` flag can be useful in your bash script to store the output of **bgdata** in a variable.

CHAPTER 5

# Advanced usage

## 5.1 Understanding the local repository

As we have already mentioned in the *package section* **bgdata** identifies each data package with a 4-level structure: *project*, *dataset*, *version* and *build*.

In the local repository, the 4-level structure is converted into a 3-level folder structure following this layout *project/dataset/version-build*.

For example, for the `hg19` version of the human genome, we set the *project* to _, the *dataset* to `genomereference`, the *version* to `hg19` and the *build* to the date used to create the package `20150724`.

If you request this package with **bgdata** (**`bgdata get _/genomereference/hg19?20150724`**), after downloading you will see that you have a local repository as:

```
|- .bgdata/
|   |
|   |- genomerefernce/
|   |   |
|   |   |- hg19-20150724/
|   |   |   |
|   |   |   |- chr1.txt
|   |   |   |- chr2.txt
|   |   |   |- ...
|   |   |   |- .downloaded
```

This structure makes easy to map the query you make with *project*, *dataset* and *version* to the folder structure.

### 5.1.1 The .downloaded

The `.downloaded` file is a file created after downloading and extracting the package used internally by **bgdata** to check whether the package is present and correct.

### 5.1.2 The .singlefile

In some data packages you will find that there is a `.singlefile` file. It contains the name of one of the files in the folder. This file, if present, is used by **bgdata** to retrieve the path to that particular file rather than the path to the folder.

**bgdata** creates this file automatically if a downloaded package contains only one file. However, some packages can use this file, even if there is more than one file, to easy the usage. For example, a tabix file is formed by a data file and and index file. However, tools using it only need to receive the path to the data file. For packages consisting on a tabix file, although they contain two files, we retrieve always the path to the data file as if that was the only file in the package.

### 5.1.3 The tag files

The `build` that is pointed by a `tag` is indicated in a file, named as the version. For example, a `tag` file for the `hg19` package mentioned above that sets the `master` tag to `20150724` build will be located in:

```
|- .bgdata/
|   |
|   |- genomereferнce/
|   |   |
|   |   |- hg19-20150724/
|   |   |
|   |   |- hg19.master
```

The `tag` file only contains a string with the `build`.

## 5.2 Cache management

**bgdata** includes some commands to manage your caches. However, keep in mind that caches are like partial copies of your local repository so adding or removing packages from your caches is as simple as copying them from the local repository or deleting.

The commands you can use with **`bgdata cache`** are:

| | |
|---|---|
| add | Add a package to the cache |
| clean | Clean everything |
| remove | Add a package to the cache |
| update | Update packages in caches |

**add** This command will copy a local package into the cache

**clean** Clean is a command to remove **everything** in the cache

**remove** This command will remove a particular build of package from the cache

**Update** Update will remove old versions of package and copy new ones. Care must be used when using this command. The flow is as follow:

- **bgdata** resolves which builds are associated with the indicated tags
- for each cache, **bgdata** gets which packages are present. If the build of that package is not in the resolved, it is deleted. The recent(s) version(s) of the packages are added to the cache.

It is important to note that if a package is not present in the cache it will not be updated.

### 5.2.1 Tags in caches

Tag files can be used in cache repositories. In fact, when you request for a particular `tag` **bgdata** looks first in the local repository and then in the caches for it.

> **Warning:** Using `tag` files in the caches is not recommended and the user must manually update the tag files.

## 5.3 Creating your own packages

### 5.3.1 Building packages

The **build** command receives the path to a folder (or even a single file) and creates a compressed data package with it. Then it uncompress it in the local repository and associates that build with the `build` tag. Thus you can use that tag (e.g. `_/genomereference/hg19?build`) for your tests.

### 5.3.2 Uploading packages to the remote

> **Warning:** This section is only for people within our group or people that have set up their own system using bgdata.

Once the package is build, it can be uploaded to the remote making use of the **upload** command.

---

**Important:** Only packages that have been previously built can be uploaded.

---

The upload process does not go through HTTP. To avoid external users to update packages to our remote repository, the upload process is just a copy of files in the network file system. Thus, it will only work for people with access to the NFS (Network File System).

If you have access, you need to edit your *configuration file* to add

```
remote_repository_upload = /path/to/remote
```

The upload process includes the creation of a metadata file for the uploaded package. This file contains, among other items, a checksum used during the download process.

## 5.4 Fixing your builds

The easiest way to fix your builds is to make it directly in your code, e.g. `bgdata get project/dataset/version?build`. However, in some cases, it is useful to fix the builds of the packages used without modifying your code. Two typical use cases are (there might be many others):

- fixing the builds for reproducibility without modifying your code. Your calls to `bgdata get project/dataset/version` will return the same build even if you add new builds.
- make a particular package point to a different *tag*. This can be useful for developing. You associate your new build to a *develop* tag and force bgdata to use the *develop* data for that package and the default for the rest.

To fix your builds without explicitly indicating that in your code, you can pass a file using the environment variable **BGDATA_BUILDS** that points to a file that sets the builds. Such file, can contain three different ways of fixing your builds:

1. Indicate a path to a file for a package in the *paths* section:

```
[paths]
project/dataset/version = /my/local/path
```

   In this case, any call to `project/dataset/version` will point to `/my/local/path`. This have no effect if the request is done indicating a *tag* or *build*.

2. Override your tags in the *builds* section:

```
[builds]
    [[project/datastet/version]]
        master = 20181105
```

   In this case, any request to the `master` *tag* of `project/datastet/version` point to `20181105` *build*. The request can be explicit (`project/dataset/version?master`) or implicit (`project/dataset/version`, when the default tag is `master`).

3. Fix the tags in the *tags* section:

```
[tags]
project/dataset/version = master
project = develop
```

   In this case, any call (that does not indicate the *build* or *tag*) of any data package under `project` will use the tag `develop` by default except the package `project/dataset/version` that will use the master. Note that this will not have any effect if you explicitly indicate the *build* or *tag*.

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search