GitHub API Bindings for Pharo

Developer Documentation

Skip Lentz

January 30, 2016

Contents

CHAPTER 1

Getting started

To start using the API, one needs to initialize the main API entrypoint, namely the `GitHub` class. One does not necessarily have to authenticate for this, but it is better to authenticate as it allows one to perform 5000 requests per hour instead of just 60 for unauthenticated users. However, if all you want is to give the bindings a quick try this can be a good option.

Authenticating can be done using either your username and password, or an access token.

An access token can be retrieved using the OAuth2 protocol, using Zinc. To do this, you might want to take a look at the documentation of Zinc SSO.

The access token can also be made manually (called a *Personal access token*), via the following link (you need to be logged in for this link to work): Personal access tokens. This is the easiest and quickest way to get started with the API. Note that you should treat this token as a password.

1.1 Initializing the main API entry point

The class `GitHub` functions as the main entry point for the API. From there, one can query either the logged in user (i.e. *you*) or a user by specifying the name, by sending the messages `GitHub>>user` and `GitHub>>user:` respectively. These messages will return an instance of `GHUser`. The following script shows how to initialize the API and request for a user:

```
| github user |

" Initialize using an access token... "
github := GitHub initializeWithAccessToken: 'f1ct10n4l4cc3sst0k3n'.
```

```
" ... or using a username and password combination. "
github := GitHub initializeWithUsername: 'JohnDoe' password: '123password'.

" ... or without authenticating yourself.
  Note that this limits the amount of requests and resources you can access. "
github := GitHub initializeAnonymously.

" Get the currently logged in user, which returns an instance of GHUser.
  This fails signalling a GHBadCredentialsError if you did not authenticate. "
user := github user.

" Get a user by their username. "
user := github user: 'MaryJane'.
```

1.2 Error handling

Errors can be handled with the regular Smalltalk syntax:

```
[ github user ]
   on: GHBadCredentialsError
   do: [ UIManager default inform: 'Incorrect username or password!' ]
```

The `GitHub` package contains several `Error` classes by default, containing a class comment explaining the conditions under which they are thrown.

Furthermore, other packages might contain `Error` classes as well, such as `GitHub-Pull-Requests` for merge failures. View the 4documentation of that package and the class comments for more detail.

1.3 Updating objects

Send `GHObject>>update` to update a domain object, and send `GHObject>>isOutdated` to test if it is outdated. The `#update` method uses `#isOutdated` internally.

1.4 Next steps

The next step is to request a user's repository. One can do this by sending `repository:` with a repository's name as argument, or sending `repositories` to get all of the repositories of the user. These methods return instances of `GHRepository` (or an `Array` of them), and can be asked several questions such as `defaultBranch`.

For more operations, such as committing, on repositories, see the following API documentation:

- 2Git Data API: Low-level operations on git concepts (objects and references).
- 3Contents API: High-level operations on files.
- 4Issues and Pull Requests API: Operations on Issues and Pull Requests.

Furthermore, there is documentation explaining the internals of the bindings: 5Internals.

CHAPTER 2

Git Data API

The low-level Git Data API allows you to create or read Git objects (blobs, trees, commits, tags) and create, delete, read and update references (refs).

This document explains how to do this using the API bindings.

2.1 Operations on low-level Git concepts

This API defines two operations on git objects (creating and reading), and four operations on git references (creating, updating, deleting and reading). The former git concepts are immutable and can therefore not be updated or deleted.

The way the API works is that one reads an object or references, changes some of its contents, and then pass it to one of the create, update or delete operations.

For example, to update a ref, one takes the ref object and change the object it is pointing to using the `GHRef>>object:` accessor. Really, the only information that this new object should have is its SHA.

So in theory, if one knows the name of the ref (for example `refs/heads/-master`) and the SHA of the object it is pointing to, one can just create new instances of GHRef and GHObject without ever having to do a request for those objects.

Here is an example of adding a blob to a tree based on the tree pointed to by the last commit on `master`:

```
1 repo := github user repository: 'FictionalRepository'.
2 head := repo getRef: 'heads/master'.
3 commit := repo getCommit: head object sha.
```

```
tree := repo getTreeRecursively: commit tree sha.

" Create a new file in-memory "
root := FileSystem memory workingDirectory.
newFile := root / 'README.md'.
newFile writeStreamDo: [ :stream |
        stream nextPutAll: 'Hello from Pharo!' ].
newEntry := GHTreeEntryWithContent
        fromFileReference: newFile
        relativeTo: root.

" Add a new blob entry. The new entries will be added on
  top of the previous entries. "
tree tree: { newEntry }.
repo createTree: tree.
```

2.2 Creating a commit.

The operations described in the previous section are tedious to use if all you want to do is create a commit. Therefore, this process is implemented in the `GHCommitBuilder` class. The builder provides several methods which simplify the process of performing a commit:

`push` This method does all the operations to create a commit, with the parameters given to the builder. Specifically, it does the following operations:

1. Take the current `HEAD` commit of the given branch (or the default branch of the repository),
2. Get the tree it points to,
3. Update the contents of this tree using the given files and create the new one,
4. Create a new commit object pointing to the new tree,
5. Update the `HEAD` reference to point to the new commit.

`directory:` This method takes a `FileReference` containing files, and stages them for the commit as if they were in the root directory of the repository.

The location of where they should be committed can be overridden by sending `baseDirectory:` afterwards with a `FileReference`. Note that this `FileReference`'s path should be contained within the reference passed to `directory:`.

An alternative to using `directory:` is to use both `files:` and `baseDirectory:`. The advantage to this is that one can omit files in the commit which are present in the reference passed to `baseDirectory:`.

`onBranch:` Specify the branch to commit on. This needs to be an existing branch, which can be created by creating a ref, described in the previous section.

`replaceTree` Let the commit replace the entire tree of the commit it is based on. Of course, this will create a new tree with just the files given. This is the only way to perform a delete operation in a commit.

CHAPTER 3

Contents API

As opposed to the 2Git Data API, this part of the binding provides a *high-level* way of reading, creating, updating and deleting files. The classes that are involved are `GHRepository` and `GHContent`. The four different use-cases are explained in the sections below.

3.1 Get file contents

To get the contents of a file, one can send the following messages to an instance of `GHRepository`:

`getContentAtPath:` Returns an instance of `GHContent` which contains the file contents of the file at the given path.

`getContentAtPath:atRef:` Same as above, but returns the file at the given version (can be a commit name, branch name or tag name).

3.2 Create a file

Use `GHRepository>>createContent:onPath:withMessage:`, with the parameters being the content of the file, the path of the file, and the commit message (all being instances of `String`).

3.3 Update a file

Use `GHContent>>updateContent:withMessage:`, with similar parameters as with the method to create files. This method updates the same instance with

the new content information, thus it is possible to send `updateContent:withMessage:` multiple times.

Moreover, it returns the `GHGitCommit` object representing the commit of the change. Thus from here it is possible to switch to using the 2Git Data API and create commits with the returned `GHGitCommit` object as parent.

3.4 Delete a file

Use `GHContent>>deleteContentWithMessage:`. Like `updateContent:withMessage:`, this method returns a `GHGitCommit` instance.

CHAPTER 4

Issues and Pull Requests API

With this API one can fetch, create and edit/manipulate Issues and Pull Requests.

4.1 Fetching an Issue or Pull Request

Issues can be fetched in two ways: by its number and by requesting a list of issues with optional parameters. The first way can be requested using `GHRepository>>issueByNumber:`, by passing it the issue number.

The second way, listing issues, allows one to specify multiple parameters. It is done by following the Builder pattern, allowing us to specify optional parameters for a result (the issue listing).

For example, one can request all issues assigned to user 'Foo', with the labels 'bug' and 'important', with this script:

```
repo issueLister
        assignedTo: 'foo';
        withLabels: #('bug' 'important');
        execute
```

For an exhaustive list of parameters, see the developer documentation on listing issues, and their corresponding methods in `GHIssueLister`.

A Pull Request can be fetched by its number as well, by using `GHRepository>>pullRequestByNumber:`.

The listing of Pull Requests like with issues is not yet implemented. However, one can use the listing of issues and then convert it into a list of `GHPullRequest`s like so:

```
| listing |
listing := repo issueLister
        " Specify optional filter parameters "
        execute.

listing
        select: #isPullRequest
        thenCollect: #asPullRequest
```

4.2 Creating an Issue or Pull Request

To create an issue, send `GHRepository>>issueCreatorWithTitle:` to a `GHRepository` instance. Similar to the listing of issues, one creates an issue by using a Builder too. The builder follows a syntax similar to that of the `GHIssueLister`. Refer to the developer documentation for all the parameters, and again their corresponding methods in `GHIssueCreator`.

The same holds for the creation of a pull request, which can be done by sending `GHRepository>>pullRequestCreatorWithTitle:head:base:` with the title of the pull request, the `head` branch specification (the one that is to be merged) and the `base` branch specification (where `head` should be merged into). Cross-repository pull requests (i.e. pull requests from forks) can be done by prefixing the username to the branch name (`username:branch`).

The API supports one other way of creating a pull request, and that is to turn an existing issue into a pull request. The bindings support this as well, and it can be done by sending `GHRepository>>createPullRequestFromIssueNumber:fromHead:toB` Once this is done, the pull request can not in turn be converted back into an issue.

4.3 Editing/Manipulating Issues and Pull Requests

Here I will explain how to manipulate an issue or pull request. By manipulate I mean any operation on an issue or pull request that one can normally do using the Web interface of GitHub.

Editing an Issue or Pull Request's properties

By sending `GHIssue>>editor` to a `GHIssue` or `GHPullRequest` instance, one gets a builder as answer with the same parameters as the `GHIssueCreator` in the previous section, with several more parameters/actions in addition. The most notable ones are `close` and `open`, which do as their name suggests.

Leaving a comment

To leave a comment on either an issue or pull request, send `addCommentWithBody`. This returns a `GHIssueComment` instance which has in turn the ability to edit and delete itself. An example:

```
" Get the first issue "
issue := repo issueLister execute first.
comment := issue addCommentWithBody: 'Hello from Pharo.'.
comment editBody: comment body , ' How are you?'.
comment delete.
```

Merging a Pull Request

To merge a Pull Request, send `GHPullRequest>>merge`. There are some other methods for merging too, which allow to set a SHA hash which should match with the current head. This way, if there has been an update since fetching the `GHPullRequest` instance, the merge will not continue.

Another method allows one to set the message of the merge commit, and yet another one allows to set both parameters.

If the merge fails, a `GHPullRequestNotMergeableError` is signalled. If the head SHA (if specified) does not match, a `GHModifiedHeadBranchError` is signalled.

4.4 The difference between Issues and Pull Requests

In short, every pull request is also an issue, and not vice-versa. This is reflected as well in the class diagram of the bindings.

Because of this, one can request a pull request by its number using the issues API. But the API will return an instance of `GHIssue`. Thus, a way is needed to test whether an issue is actually a pull request, or if it's just an ordinary issue. This is provided by `GHIssue>>isPullRequest`.

Furthermore, an existing `GHIssue` can be converted into a `GHPullRequest` by sending `GHIssue>>asPullRequest`. Do note that this will perform a request, since a pull request contains extra information not present in an issue. If the `GHIssue` is not a pull request (`#isPullRequest` returns false), this results in a `GHNotAPullRequestError`.

CHAPTER 5

Internals

This doc explains more on the internals of the API.

5.1 Requesters

A Requester is any API object which can make further requests. In the case of `GHContent` for example, one can send `GHContent>>updateContent:withMessage:`. Requesters are users of the `GHTRequester` trait.

Not all API objects are Requesters. A good example are the subclasses of `GHGitObject` and `GHRef`, which are mutable for the purpose of passing them as parameters (more on the Git Data API 2here).

Requesters can be more easily seen as an extension wrapper around Zinc's `ZnClient` class. Like `ZnClient`, Requesters are stateful. Requesters expand Zinc with functionality for the API. In short, these functionalities are:

- Parsing the JSON representation to an API object
- Error handling
- Conditional requests

The following sections will go more in-depth on them.

5.2 Response Parsing

The parsing of the JSON is done with the excellent NeoJSON library (read this doc for an introduction). In NeoJSON there is the concept of a mapping, which defines how to read from (and write to) a JSON representation into an instantiated object. The mapping of an object can be one of `ObjectMapping` or

`CustomMapping`. The first is one that maps to a class' properties (in our case just instance variables), the latter is one that allows for a custom definition of a mapping (how the value should be interpreted).

The API bindings provide a trait, `GHTMappedToJSON`, which contains only class-side methods related to NeoJSON mappings. The trait includes the mapping of URLs (instances of `ZnUrl`) and `DateAndTime` instance variables. Users of this trait can then extend the mappings with custom ones for other types.

Generating instance variables, accessors and mapping definitions

When creating a new API object, one might want to generate some of the instance variables, accessors and mapping definitions automatically.

This can be done using `GHRelObject class>>generateInstanceVariablesAndMethodsWithA` which when provided an array of API keys (e.g. `created_at`, `username`), generates instance variables and accessors for them.

Furthermore, API keys with the suffix `_url` are automatically set to be mapped as `ZnUrl`, and those ending with `_at` will be mapped to `DateAndTime` instances.

5.3 Error Handling

The handling of errors is done before parsing the response. The error handlers are defined as `BlockClosures` which take a response as argument. They are stored in a `Dictionary` with as their key the HTTP integer error code (e.g. `404`). By default they trigger an `Error`, which allows one to use the regular Smalltalk syntax for handling them:

```
1 [ github user ]
2    on: GHBadCredentialsError
3    do: [ UIManager default inform: 'Incorrect username or password!' ]
```

5.4 Conditional Requests

Conditional requests allow one to test if a resource was modified. If it was modified, one gets back the new resource as JSON. If not, the response will be `304 Not Modified` without any content (for reducing bandwidth).

The check of whether or not a resource has changed is done with a hash value called an *ETag*. Any Requester has access to these ETags using the `GHTRequester>>urlToETag` accessor.

A conditional request is made by sending `GHTRequester>>conditionalGet:` with an URL. It uses the `#urlToETag` `Dictionary` to get the ETag when a new request is made. However it is probably not necessary to use this method directly.

Often one wants to ask to a domain object whether its remote resource has changed. This is provided by `GHObject>>isOutdated`, which performs a conditional request (if necessary) with its own url as argument.

Even more often one wants to ask a domain object to update itself if it changed, which can be done using `GHObject>>update`. This method uses `#isOutdated` internally, and if it returns true it copies all the instance variables of the response to itself.