CLI Tools

The Skara command-line interface (CLI) tools enable a CLI driven workflow where reviews are made either via the mailing lists or in a web browser using an external Git source code hosting provider's web application. The following CLI tools are currently available as part of project Skara:

- `git-jcheck` - a backwards compatible Git port of jcheck
- `git-webrev` - a backwards compatible Git port of webrev
- `git-defpath` - a backwards compatible Git port of defpath
- `git-trees` - a backwards compatible Git port of trees
- `git-fork` - fork a project on an external Git source code hosting provider to your personal space and optionally clone it
- `git-sync` - sync the personal fork of the project with the current state of the upstream repository
- `git-backport` - fetch a commit from a remote repository and apply it on top of the current branch
- `git-pr` - interact with pull requests for a project on an external Git source code hosting provider
- `git-info` - show OpenJDK information about commits, e.g. issue links, authors, contributors, etc.
- `git-token` - interact with a Git credential manager for handling personal access tokens
- `git-translate` - translate between Mercurial and Git hashes
- `git-publish` - publish a local branch to a remote repository
- `git-proxy` - proxy all network traffic from a Git command through a HTTP(S) proxy
- `git-skara` - learn about and update the Skara CLI tool

All above CLI tools support multiple external Git source code hosting providers.

It is **not** necessary to use the Skara CLI tools to contribute changes to OpenJDK. Contributors that prefer to use e.g. desktop applications, web browsers and/or IDEs that integrate with applicable external Git source code hosting providers are free to do so. The Skara team's recommended setup is using the Git CLI client and a web browser.

Table of Contents

Installing

To install the Skara tooling, simply clone the Skara repository and include the Skara Git configuration file:

```
$ git clone https://github.com/openjdk/skara
$ git config --global include.path "$PWD/skara/skara.gitconfig"
```

If you are running on an x64 system using Linux, Macos or Windows, the Skara tooling will bootstrap itself the first time you use any of the Skara commands. For other platforms you will need to explicitly provide a JDK 16 or later and run the build directly:

```
$ JAVA_HOME=/path/to/jdk-16/or/later bash gradlew
```

To check that everything works run `git skara help`:

```
$ git skara help
```

Note: if your computer is behind a HTTP(S) proxy, ensure that you have set the `HTTPS_PROXY` environment variable correctly.

Note: installing skara more than once can cause issues. If `git config --get-all include.path` returns more than one line, the skara bootstrap mechanism will get confused. Either make sure to only have one installation, or edit that line to read `grep 'skara.gitconfig' | tail -1` assuming the last one is the right one.

For additional ways to install the Skara CLI tooling, see project Skara's README.

Updating

To update the Skara tooling run `git skara update`:

```
$ git skara update
```

The update command pulls eventual updates and rebuild the tooling if necessary. *Note:* if your computer is behind a HTTP(S) proxy, ensure that you have set the `HTTPS_PROXY` environment variable correctly.

If you are using system other than Linux, Macos, or Windows x64, you need to provide a JDK 16 or later and run the build directly as described above after using git to update the Skara repository.

If the update command for some reason isn't working or you just want to manually retrace the steps you can each step manually like this:

```
$ git pull
$ bash gradlew
```

Personal Access Token

Some of the Skara tools requires a *personal access token* (PAT) to authenticate against an external Git source code hosting provider's API. A personal access token is a like a password that has limited capabilities, it can only be used to successfully authenticate and perform certain limited actions. The following Skara tools requires a personal access token:

- `git-fork`
- `git-pr`
- `git-token`

Note: if you do not intend to use the above three tools, then there is no need to set up a personal access token and you can skip this section. All the other tools described in the beginning of this document works fine without a personal access token. If you do wish to make use of the above three tools, then please read on.

Configuring a personal access token consists of two steps:

1. Set up a credential manager for securely storing the personal access token locally on your computer
2. Generate the personal access token and store it in the credential manager

The way to carry out the above two steps differs depending on the operating system you use, please follow the instructions below suitable for your operating system.

Windows

Credential Manager

If you installed Git via Git for Windows and have a recent version, then you already have a credential manager from Microsoft installed (it is bundled with Git for Windows, but make sure to pick it during installation). If you installed Git via some other mechanism, then you must first install Microsoft's Git Credential Manager. If you have an older version of Git for Windows and using the deprecated Git Credential Manager for Windows, you may need to configure git to use the credential manager like this:

```
$ git config --global credential.helper manager
```

Personal Access Token

To generate a a personal access token on GitHub go to https://github.com/settings/tokens and and click on "Generate new token". You only need to select the "repo" scope (permission). After you have generated your personal access token, store it in Keychain using `git token store:`

```
$ git token store https://github.com
Username: <insert your Github username>
Password: <insert your "Personal Access Token", not your GitHub password>
```

macOS

Credential Manager

macOS already comes with a password manager in the form of Keychain and Git for macOS is configured out of the box to use Keychain as a credential manager, there is no need to configure anything.

Personal Access Token

To generate a a personal access token on GitHub go to https://github.com/settings/tokens and and click on "Generate new token". You only need to select the "repo" scope (permission). After you have generated your personal access token, store it in Keychain using `git token store:`

```
$ git token store https://github.com
Username: <insert your Github username>
Password: <insert your "Personal Access Token", not your GitHub password>
```

GNU/Linux

The credential manager you will use on GNU/Linux to securely store the personal access token depends on your desktop environment. If you are using a desktop environment with support for GNOME Keyring, then follow the instructions in the GNOME Keyring section. If you are using a GNU/Linux installation *without* a desktop environment (e.g. when using SSH to connect to a server) or a desktop environment that does not support the GNOME Keyring (e.g. XFCE, KDE, i3), then you need to pick a credential manager that suits your security and usability needs. The following sections will present four common choices for storing personal access tokens when you are unable to use GNOME Keyring:

- GnuPG
- age
- pass
- plain text file (insecure)

GNOME Keyring

Credential Manager

On GNU/Linux installations that feature the GNOME Keyring the recommended setup is to use libsecret and the "libsecret credential helper" in order to use GNOME Keyring as the credential manager. Please follow the instructions for setting up libsecret for the GNU/Linux distribution you are using.

Fedora

Fedora 30, 31 and 32 comes with libsecret and GNOME Keyring installed by default. When you install the git package you also get the libsecret credential helper automatically installed. To configure git to use the libsecret credential helper run the following command:

```
$ git config --global credential.helper /usr/libexec/git-core/git-credential-libsecret
```

If you want to have a graphical utility to inspect the GNOME Keyring we recommend that you install GNOME Seahorse:

```
sudo dnf install seahorse
```

Ubuntu

Ubuntu 19.10 and 18.04.4 (LTS) comes with libsecret and GNOME Keyring installed by default. Unfortunately even if you install the Git package you will not get a binary version of the libsecret credential helper installed (you only get the source). This means you have to compile the libsecret credential helper yourself. Compile the libsecret credential with the following two commands:

```
$ sudo apt install libsecret-1-dev
$ sudo make --directory=/usr/share/doc/git/contrib/credential/libsecret
```

Once you have compiled the libsecret credential helper you must configure Git to use it:

```
$ git config --global credential.helper /usr/share/doc/git/contrib/credential/libsecret/git-credential-libsecret
```

If you want to have a graphical utility to inspect the GNOME Keyring we recommend that you install GNOME Seahorse:

```
$ sudo apt install seahorse
```

Personal Access Token

To generate a personal access token on GitHub go to https://github.com/settings/tokens and click on "Generate new token". You only need to select the "repo" scope (permission). After you have generated your token, store it in the GNOME Keyring using `git token store`:

```
$ git token store https://github.com
Username: <insert your Github username>
Password: <insert your "Personal Access Token", not your GitHub password>
```

GnuPG

You can use GnuPG (GPG) to store your personal access token encrypted in a file. You will first have to store your GitHub username in the Git configuration file by running the following command (replace <USERNAME> with your GitHub username):

```
$ git config --global 'credential.https://github.com.username' <USERNAME>
```

The next step is to generate a personal access token. Go to https://github.com/settings/tokens and click on "Generate new token". You only need to select the "repo" scope (permission). After you have generated your token, use GPG to encrypt it and store it in a file. The personal access token can be encrypted either using a GPG key *or* using a passphrase. If you have a GPG key you probably already know how to encrypt text with it, so we will only cover encryption using a passphrase here. To encrypt the personal access with a passphrase and store it in a file, run the following command (replacing <PAT> with the personal access token you just generated):

```
$ echo '<PAT>' | gpg --symmetric --output ~/github-pat.gpg
Enter passphrase:
Repeat passphrase:
```

Finally you must configure Git to decrypt and read the personal access from the file `~/github-pat.gpg` when credentials are needed for https://github.com. This is done by the following command:

```
$ git config --global 'credential.https://github.com.helper' '!f() { test $1 = get && echo password=`gpg --
decrypt ~/github-pat.gpg`; }; f'
```

age

You can use age to store your personal access token encrypted in a file. You will first have to store your GitHub username in the Git configuration file by running the following command (replace <USERNAME> with your GitHub username):

```
$ git config --global 'credential.https://github.com.username' <USERNAME>
```

The next step is to generate a personal access token. Go to https://github.com/settings/tokens and click on "Generate new token". You only need to select the "repo" scope (permission). After you have generated your token, use age to encrypt it and store it in a file. To encrypt the personal access token using a passphrase and storing the result in a file, run the following command (replacing <PAT> with your personal access token):

```
$ echo '<PAT>' | age --passphrase > ~/github-pat.age
```

Finally you must configure Git to decrypt and read the personal access token from the file `~/github-pat.age` when credentials are needed for https://github.com. This is done by the following command:

```
$ git config --global 'credential.https://github.com.helper' '!f() { test $1 = get && echo password=`age --
decrypt ~/github-pat.age`; }; f'
```

pass

You can use pass to store your personal access token encrypted in a file. You will first have to store your GitHub username in the Git configuration file by running the following command (replace <USERNAME> with your GitHub username):

```
$ git config --global 'credential.https://github.com.username' <USERNAME>
```

The next step is to generate a personal access token. Go to https://github.com/settings/tokens and click on "Generate new token". You only need to select the "repo" scope (permission). After you have generated your token, use pass to store the personal access token securely:

```
$ pass insert github/pat
Enter password for github/pat: <insert your "Personal Access Token", not your GitHub password>
```

Finally you must configure Git to read the personal access token when credentials are needed for https://github.com. This is done by the following command:

```
$ git config --global 'credential.https://github.com.helper' '!f() { test $1 = get && echo password=`pass github
/pat`; }; f'
```

Plain text file (insecure)

This is **not** as secure as storing the personal access token encrypted. Any person or program who can read `~/.git-credentials` will be able to read your personal access token and impersonate you.

An insecure way to store your personal access token is to configure Git to store the personal access token unencrypted in the file `~/.git-credentials`. You can configure Git do to this by running the following command:

```
$ git config --global credential.helper store
```

To generate a personal access token on GitHub go to https://github.com/settings/tokens and click on "Generate new token". You only need to select the "repo" scope (permission). After you have generated your token, store it in `~/.git-credentials` by running `git token store`:

```
$ git token store https://github.com
Username: <insert your Github username>
Password: <insert your "Personal Access Token", not your GitHub password>
```

Commands

Please see the documentation for each tool on the tool's individual wiki page:

- git-defpath
- git-fork
- git-hg-export
- git-info
- git-jcheck
- git-proxy
- git-publish
- git-skara
- git-sync
- git-token
- git-translate
- git-trees
- git-webrev

Overview

The following sections contains examples on how to use the Skara CLI tools. For more detailed information on how to use a certain tool, see the documentation for that tool.

Creating a personal fork

To create a personal fork of an upstream repository, run the command `git fork <URL>`. For example, to create a personal fork of the jdk repository, run:

```
$ git fork https://github.com/openjdk/jdk
```

The command `git fork` will also clone your personal fork to a local repository on your computer.

Publishing a local branch

To publish a local branch to a remote repository, run the following command:

```
$ git publish
```

Creating a pull request

To create a pull request first create a personal fork, then create a local branch in the local clone of your personal fork. Make changes to a number of files, then create a commit. Publish your local branch and then create a pull request from your published branch:

```
$ git pr create
```

Notes:

- If you want to run jcheck on your changes *before* the pull request is created, pass the flag `--jcheck`
- If you want the local branch to be published automatically, pass the flag `--publish`

Listing pull requests

To list the open pull requests for a repository, run the command `git pr list` in a local clone of your personal fork:

```
$ git pr list
```

Notes:

- You can filter the listed pull requests by passing additional flags such `--assignees=<USERNAMES>`, `--authors=<USERNAMES>`, `--labels=<LABELS>`
- You can select the columns to show by passing the `--columns` flag, for example `--columns=id,title`

Setting properties of a pull request

To set properties of a pull request, run the command `git pr set`:

```
$ git pr set
```

Examples:

- To set the title of a pull request, run `git pr set --title <TITLE>`
- To set the body of a pull request, run `git pr set --body <BODY>`
- To close a pull request, run `git pr set --closed`

Integrating a pull request

To integrate a pull request that you have created, run the command `git pr integrate`:

```
$ git pr integrate
```

Notes:

- If you find yourself typing `git pr integrate` a lot, you might want to create the alias "integrate":

  ```
  $ git config --global alias.integrate 'pr integrate'
  ```

  You can then just run `git integrate` to integrate a pull request.

Syncing a personal fork

To sync your personal fork with the upstream repository it was created from, run the command `git sync`:

```
$ git sync
```

Notes:

- To sync your personal fork *and* your local repository, pass the flag `--fast-forward`
- To sync only a subset of branches, pass the flag `--branches=<BRANCHES>`

Showing information about a commit

To show additional information about a commit, such as a link to a pull request or JBS issue, run the command `git info`:

```
$ git info
```

Notes:

- You can show a subset of the fields by passing flags, for example `--author`, `--issues`, `--review`, `--sponsor` etc.