JEP 369: Migrate to GitHub

Authors	Erik Duveblad, Joe Darcy
Owner	Joe Darcy
Type	Infrastructure
Scope	JDK
Status	Closed / Delivered
Release	16
Component	infrastructure
Discussion	discuss at openjdk dot java dot net
Effort	L
Duration	M
Relates to	JEP 296: Consolidate the JDK Forest into a Single Repository JEP 357: Migrate from Mercurial to Git
Endorsed by	Mark Reinhold
Created	2019/11/07 18:54
Updated	2021/01/15 04:03
Issue	8233813

Summary

Host the OpenJDK Community's Git repositories on GitHub. In concert with JEP 357 (Migrate from Mercurial to Git), this would migrate all single-repository OpenJDK Projects to GitHub, including both JDK feature releases and JDK update releases for versions 11 and later.

Goals

- Host all OpenJDK Git repositories at https://github.com/openjdk/.
- Run pre-commit checks (jcheck) before every push.
- Integrate existing OpenJDK services.
- Enable multiple ways of interacting with GitHub.
- Ensure that workflows structurally similar to the existing e-mail and webrev based workflows are supported.
- Preserve and archive all metadata.
- Ensure that OpenJDK Community can always move to a different source-code hosting provider.
- Do not require developers to install OpenJDK specific tools in order to contribute.
- Do not change the OpenJDK Bylaws.
- Do not change the OpenJDK Census.

Non-Goals

It is not a goal to change the OpenJDK Community's issue tracker, wiki, or any other existing infrastructure.

Success Metrics

- Significantly faster clone and pull times
- Better availability (uptime) of repositories
- Possible to interact with repositories on GitHub via OpenJDK mailing lists
- Possible to interact with repositories on GitHub via command-line tools
- Possible to interact with repositories on GitHub via web browsers

Motivation

The motivation for this JEP is split into two sections. First we cover why it would benefit the OpenJDK Community to make use of an external source-code hosting provider, and then we explain why GitHub is the best choice of provider at this time.

Why an external source-code hosting provider?

An external source-code hosting provider is a source-code repository service that is not implemented and managed by contributors in the OpenJDK Community. Examples of external providers are BitBucket, GitLab, Phacility, SourceForge and GitHub.

There are three primary reasons for the OpenJDK Community to use an external source-code hosting provider:

- *Performance*. Many, if not all, providers have excellent performance, not only with regard to network performance but also when it comes to availability, i.e., uptime. For the OpenJDK Community this would result in significantly faster clone and pull times, and more highly-available source-code repositories.
- *API*. A technical reason to host OpenJDK repositories on a source-code hosting platform is to gain access to web APIs that enable programs to interact with developers on the platform. Although not impossible to achieve today by interacting with developers over email, it is considerably harder to implement programs that interpret free-form text in emails compared to using a structured API. Allowing programs to participate in the review process enables powerful automation; see the Description section for several examples.
- *Expanded community.* The largest providers would also offer the OpenJDK Community the opportunity to tap into large existing communities of developers and potential contributors. If a developer already has an account on a provider then few additional steps are required in order to contribute to OpenJDK. Almost all open-source projects in the larger Java community are hosted on an external providers, including several OpenJDK distributions. This can foster an even closer collaboration if OpenJDK repositories are also hosted on the same provider.

One drawback and risk of using an external provider is the possibility that the provider might shut down, or for some other reason make the source code and related materials inaccessible. See the Risks and Assumptions section for how this risk can be handled and mitigated.

Another consideration is the effect that a move to an external provider would have on the workflow of existing contributors. See the Description section for how multiple workflows can be supported with the help of the APIs from source-code hosting platforms, including how almost all of the OpenJDK Community's existing workflow can be preserved.

Why GitHub?

The motivation for choosing GitHub is that is excels at all three primary reasons for choosing an external provider. GitHub's performance is as good as or superior to other providers, it is the world's largest source-code hosting service (50 million users as of May 2020), and it has one of the most extensive APIs.

GitHub's extensive API has enabled support for GitHub in many tools including text editors, IDEs, command-line tools, and graphical desktop clients. The following text editors support GitHub (i.e., a developer can create, review, and comment upon pull requests directly from within the text editor):

- Emacs (magit plugin + forge plugin)
- VS Code (GitHub Pull Request plugin)
- Atom (builtin)

The following integrated development environments (IDEs) support interacting with pull requests on GitHub:

- IntelliJ (builtin)
- Eclipse (GitHub Mylyn Connector)
- Visual Studio (GitHub Extension)

There is also a command-line client in the form of hub (open source) and several graphical desktop clients, e.g., SourceTree, Tower and GitHub Desktop (open source).

There are many other excellent source-code hosting providers. See the Risk and Assumptions section for why this is important and how this enables us to recommend GitHub.

Description

The interesting part of moving to an external source-code hosting provider would *not* be in uploading the actual repositories to the service; that is trivial given the work in JEP 357 (Migrate from Mercurial to Git).
What is interesting is how the move to an external provider would affect the workflow of OpenJDK contributors.

Today, OpenJDK contributors collaborate via mailing lists, they push changes to Mercurial repositories, they test changes via the jdk/submit service, and they file bug reports via the JDK Bug System (JBS). Contributors can also make use of several command-line interface (CLI) tools, primarily jcheck, webrev, and defpath. This is a workflow that many experienced contributors enjoy and find productive. It's therefore critical to preserve as much of this workflow as possible if we move to an external provider.

The workflow on GitHub and other popular hosting providers employs the concept of a *pull request* (PR), which on a high level is similar to the request-for-review (RFR) emails that OpenJDK contributors use today. A contributor on GitHub creates a PR from a source branch towards a target branch in particular repository (the source and target branch do not have to reside in the same repository). The PR is then reviewed by other contributors and additional commits are made to the source branch in response to reviewer feedback. Once the reviewers are happy with the changes, the PR is merged into the target branch.

This JEP proposes to support multiple workflows, using the tooling developed in Project Skara. Contributors would be able to choose whichever workflow suits them best:

- CLI + mailing-list based (similar to the current workflow)
- Web-browser based (via the GitHub website)
- Text-editor and IDE based (via plugins and GitHub support in text editors and IDEs)
- CLI based (via custom CLI tooling)

It is of course possible to mix and match the different workflows. With an open-source community as large and diverse as OpenJDK, the only common thing about individual contributors' workflows is that they differ. Before describing the various tools and services, we start with an example of the workflow for incorporating a new change. (Workflows for more specialized tasks such as syncing between repositories or adding tags will be discussed in future revisions of this JEP.)

Workflow

Common for all four workflows is that every change starts with a pull request (PR). The PR can be created in various ways -- from the CLI, a web browser, a text editor, or an IDE. However, creating a PR requires an account on GitHub. For contributors not wishing to create an account on GitHub, we would continue to accept patches sent to the OpenJDK mailing lists. Such patches would, however, require a contributor with a GitHub account to create a PR, similar to how many contributors today help newcomers to create and upload webrevs to cr.

After a PR is created, the commits in the PR are analyzed by the jcheck commit-analysis tool, which is run as a server-side program (a so-called "bot"). The jcheck bot uses the external provider's API to present eventual issues as comments or errors, depending on the capabilities of the provider's API. On GitHub the jcheck bot specifically makes use of the Checks API to produce informative error messages. The jcheck bot is configurable via the `.jcheck/config` configuration file in a repository. If configured, one of the checks ensures that the proposed change has a corresponding issue in JBS. In that case the bot ensures that the title of the PR corresponds to the title of the JBS issue, similar to the titles of RFR emails. If the PR contains a JBS issue as its title then the jbs bot adds a link to the PR in the corresponding JBS issue.

Once the PR passes jcheck then it gets the label "`rfr`". This highlights to potential reviewers that the PR now is ready for review. (There is no point in reviewers engaging with a change that doesn't even pass jcheck.) At this time the PR is also automatically labeled according to the areas of the source code that are changed by the commits in the PR. For example, a PR to the `master` branch in the jdk repository with changes in both the make directory and the `src/hotspot` directory gets the labels `build-dev` and `hotspot-dev`. An automatically generated RFR email is sent to the mailing lists `build-dev@openjdk.java.net` and `hotspot-dev@openjdk.java.net` and to any other lists specified by the PR author. The RFR email contains the title and body of the PR, an automatically generated summary of the commits in the PR, and links to automatically generated webrevs.

Reviewers can now discuss the changes in the PR, using multiple workflows:

- By replying to the RFR email sent to the mailing list(s), in which case the contents of replies are copied into the PR on GitHub (no GitHub account is required);
- By using the review tool on https://github.com/ via a web browser;
- By using the review tool in various text editors and IDEs that integrate with GitHub;
- By using a graphical desktop application that integrates with GitHub; or
- By using the Skara CLI tools.

Any comment made in *any* of the workflows is reflected in *all* of the workflows. One reviewer can add a comment via the mailing list, another via the web browser, and a third via the command-line and they will all see each others' comments.

If the PR author pushes additional commits to the source branch based on feedback from reviewers then a bot sends a reply to the RFR email, including an automatic summary of the new commit(s) and automatically generated incremental and full webrevs.

As reviewers are satisfied with the changes, they mark the PR as reviewed. This can be done by:

- Using a web browser and https://github.com/,
- Using `git pr approve` from the command line,
- Using a text editor and/or IDE with GitHub integration, or
- Using a graphical desktop tool with GitHub integration.

Once all reviewers are satisfied, the author of the PR can finally integrate the PR. They do this by adding a comment with the exact content `/integrate`, after which a number of things happen:

1. A bot squashes (collapses) all commits into one commit.
2. If necessary, a bot rebases the squashed commit on top of the target branch (usually `master`).
3. A bot constructs a commit message for the final commit, which includes:
   - The title of the PR as the title for the commit message,
   - A body based on a comment from the PR starting with with the word `/summary`,
   - All reviewers' OpenJDK usernames added in a `Reviewed-by:` trailer, and
   - All co-authors added in `Co-authored-by:` trailers.
4. A bot pushes the resulting commit to the target branch (usually `master`).

The author can preview the result of issuing the `/integrate` command, including the commit message, before actually integrating the PR.

If the author of the PR is not an OpenJDK Committer then the label `sponsor` is added after the author has issued the `/integrate` command in a comment. An OpenJDK committer must then type the `/sponsor` command in a new comment to integrate the PR.

Permissions and roles

As noted earlier we do not propose to change the OpenJDK Census, nor would OpenJDK Contributors' roles and permissions be modeled on an external source-code hosting platform. Instead the server-side tools ("bots") maintain a mapping of OpenJDK Contributors' GitHub user ids (not usernames, which are unstable) to OpenJDK usernames. The result is that the bots can check if, e.g., the author of a PR is an OpenJDK Author or that the reviewer of a PR is an OpenJDK Reviewer. This model is independent of external provider. It also means that there is no need to duplicate the OpenJDK Census data on an external provider.

Tools

Contributors in the Skara project have developed various tools in support of this JEP, both CLI tools that run locally on a contributor's computer and server-side tools ("bots") that run on providers' servers. The CLI tools are primarily meant to assist contributors wishing to interact with an external provider from the command line. These are:

- `git-fork`: fork a project on an external provider and clone it
- `git-pr`: update, approve, fetch, show, etc., a pull request
- `git-sync`: sync branches from an upstream repository
- `git-publish`: publish a local branch to a remote repository
- `git-info`: extract OpenJDK-specific information from a commit message
- `git-token`: interact with a Git credential manager for handling tokens
- `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

The following CLI tools were already made available in support of JEP 357 (Migrate from Mercurial to Git):

- `git-jcheck`: run jcheck locally
- `git-webrev`: create a webrev
- `git-defpath`: set up push and pull paths
- `git-translate`: translate between hg and git hashes

The bots that assist contributors are:

- `pr`: checks, labels, and integrates pull requests
- `ml-bridge`: bridges comments between mailing lists and source-code hosting services
- `notify`: sends notifications to mailing lists after pushes
- `archiver`: archives all PR metadata in JSON format in a Git repository
- `forwarder`: forwards pushed commits to other repositories (e.g., a sandbox)
- `jbs`: updates JBS (similar to the existing "hgupdater" service)
- `submit`: an example of a test service implemented as a bot

Services

In addition to the bots, there are two services to aid contributors and reviewers:

- The *OCA service* relieves reviewers of the burden of checking whether contributors have signed the Oracle Contributor Agreement.
- The *test service* is a generalized version of the jdk/submit repository. It allows a contributor to issue a `/test` request in a PR comment, to which one or more test services can respond. An simple example of a test service is already available in the form of a submit bot.

Alternatives

- Keep using Mercurial and the existing OpenJDK workflow. (For repos as large as those for the JDK, we do not expect it to be practical to use `hg-git` or similar tools to retain a client-side Mercurial version of a server-side Git master repo.)
- Use GitLab EE as the external source-code hosting provider.
- Use BitBucket as the external source-code hosting provider.

Testing

- Almost all of the scenarios for the workflows have been implemented as integration tests in the Skara project.
- The Skara project has used the proposed workflow for its own code for a long period of time.
- Several other OpenJDK Projects have moved their repositories to GitHub on an evaluation basis: OpenJFX, Loom, Mobile, OpenJMC, Panama, Metropolis, Valhalla, Amber, Tsan, ZGC, Lanai and some of the Code Tools projects. These Projects have provided real-world testing of the tools, bots, and services to ensure that further Project transitions can be done with a low amount of friction.

Risks and Assumptions

The primary risk of switching to an external source-code hosting provider is that the OpenJDK Community may become dependent upon that provider. The version-control data itself will always be independent of provider, due to distributed nature of Git. There is, however, a risk that metadata such as code-review comments could become locked in to a particular provider. There is also a risk that tooling and workflows could become dependent upon a particular provider, so that changing providers becomes impossible due to assumptions made in the tooling.

Mitigating these risks has been a primary focus of the Skara project. The following design decisions ensure that critical metadata would not be locked in to any particular provider:

- All discussions in pull requests are replicated to the corresponding OpenJDK mailing lists.
- All discussions in pull requests are archived in two formats: mbox (for human consumption) and JSON (for software consumption).
- Push notifications are sent to the corresponding `*-changes@openjdk.java.net` mailing lists, thereby avoiding a dependence upon a provider's RSS feeds.
- The OpenJDK Census is used for user organization and privilege levels, thereby avoiding a dependence upon a provider's user-organization tools.
- The domain https://git.openjdk.java.net/ redirects to the OpenJDK Community's current source-code hosting provider. Source-code URLs recorded in JBS issues and mailing-list messages use this domain rather than the domain of the current provider.

In order to prevent the Skara tooling from depending upon a particular provider's API, support for multiple external providers has been a strict requirement from the beginning. All of the tooling is also required to work with the open-source GitLab Community Edition (GitLab CE).

Dependencies

- JEP 296: Consolidate the JDK Forest into a Single Repository
- JEP 357: Migrate from Mercurial to Git